V-Link32 Voice Development Kit
User's Manual

Version 1.0

Trademarks：

IBM and PC-AT are the trademarks of International Business Machines Corporation. IBM PC and PC/XT are the trademarks of International Business Machines Corporation. MS-DOS, Windows and Windows NT are the trademarks of Microsoft Corporation.

All referred trademarks not listed above are the trademarks of their respective companies.

Copyright © 2002, ELETECH ENTERPRISE CO., LTD.
All Rights Reserved.

Table of Contents

Installation 1
- Install V-Link32 Voice Development Kit 1
- Hardware Settings 6
- Waveform Driver Supported 6

Introduction 9
Function Summary 11
Programming Models 13
- Synchronous Programming 13
- Asynchronous Programming 13
- Polling Model (Asynchronous) 13
- Function Callback Model (Asynchronous) 14
- Window Callback Model (Asynchronous) 15

Event Management 19
- Termination Event 19
- CST Event 19
  - Polling Model 20
  - Function Callback Model 20
  - Window Callback Model 21

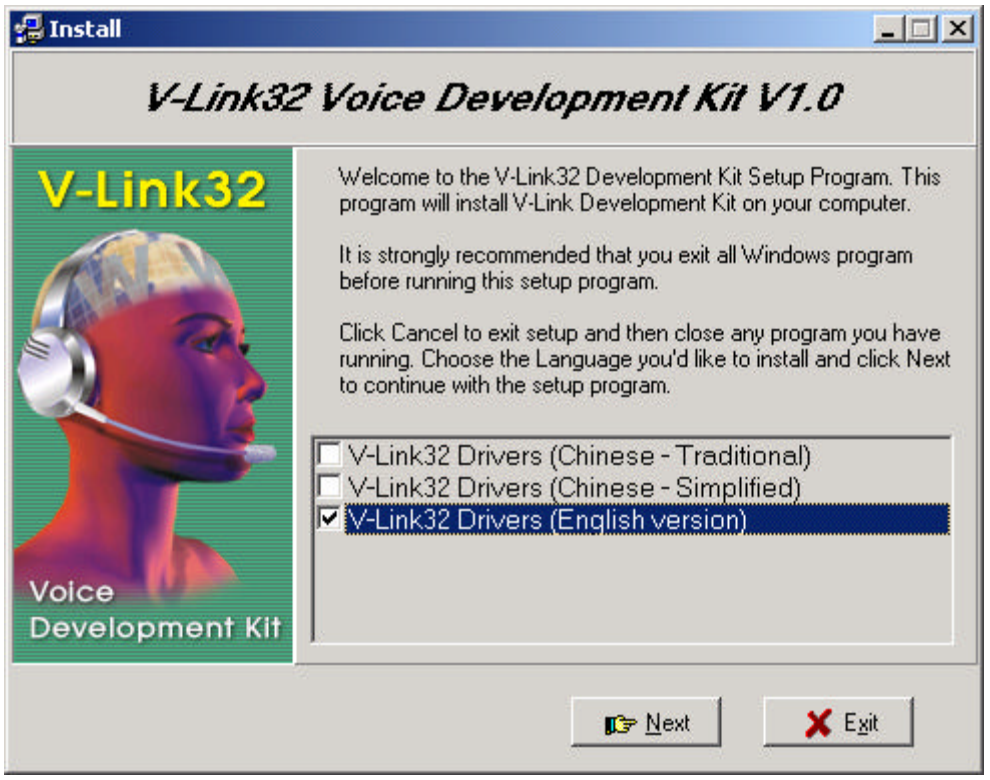
Function Return Codes 25
Function Description 27
- adsiCAS 28
- adsiCheckSum 31
- adsiRecvFrame 32
- adsiSetParam 35
- adsiXmitFrame 37
- anaGetXmitSlot 40
- anaListen 41
- anaUnlisten 43
- vocAddToConference 44
- vocBreakConference 45
- vocClearDT 46
- vocCloseChn 47
- vocCutWaveFile 48
- vocDelFromConference 49
- vocDial 50
- vocEnumChn 53
- vocFlashHook 54
- vocGetCallerID 56
- vocGetCAR 58
- vocGetChnCaps 60
- vocGetChnID 61
- vocGetChnIO 62
- vocGetConfGTD 63
- vocGetConfVol 64
- vocGetCurPos 65
- vocGetDeviceID 66
- vocGetDT 67
- vocGetGTD 70
- vocGetLastCST 71
- vocGetLastErr 72
- vocGetLastEvent 73
- vocGetLastTerm 74

vocGetSerialNo 75
vocGetTermDT 76
vocGetVolume 77
vocGetXmitSlot 78
vocInitDriver 79
vocIsLineConnect 80
vocListen 81
vocMakeConference 83
vocMonitorChn 86
vocOpenChn 88
vocPlayFile 90
vocPlayTone 93
vocPutSignal 95
vocQueryConfGTD 97
vocQueryGTD 98
vocReadDT 99
vocReadOEMVersion 100
vocRecordFile 101
vocSetChnIO 104
vocSetChnParam 105
vocSetConfVol 107
vocSetCSTMask 108
vocSetEventCallback 109
vocSetGTDMask 110
vocSetHook 111
vocSetVolume 113
vocStopChn 114
vocSwitchFax 116
vocUnlisten 118
vocWaitConfEvent 119
vocWaitCST 120
vocWaitEvent 122
vocWaitRing 124
vocWaitRingEx 125
CHNMON32 Program 127
monShowCST 128
monShowMSG 129
Application Notes 130
Caller ID 132
Overview 132
FSK 132
DTMF 133
Enabling the Caller ID feature 134
Example 134
ADSI (Analog Display Services Interface) 136
Overview 136
Frame Format 136
FSK Modulation and Demodulation 138
Example 138
Frequently Asked Questions 143

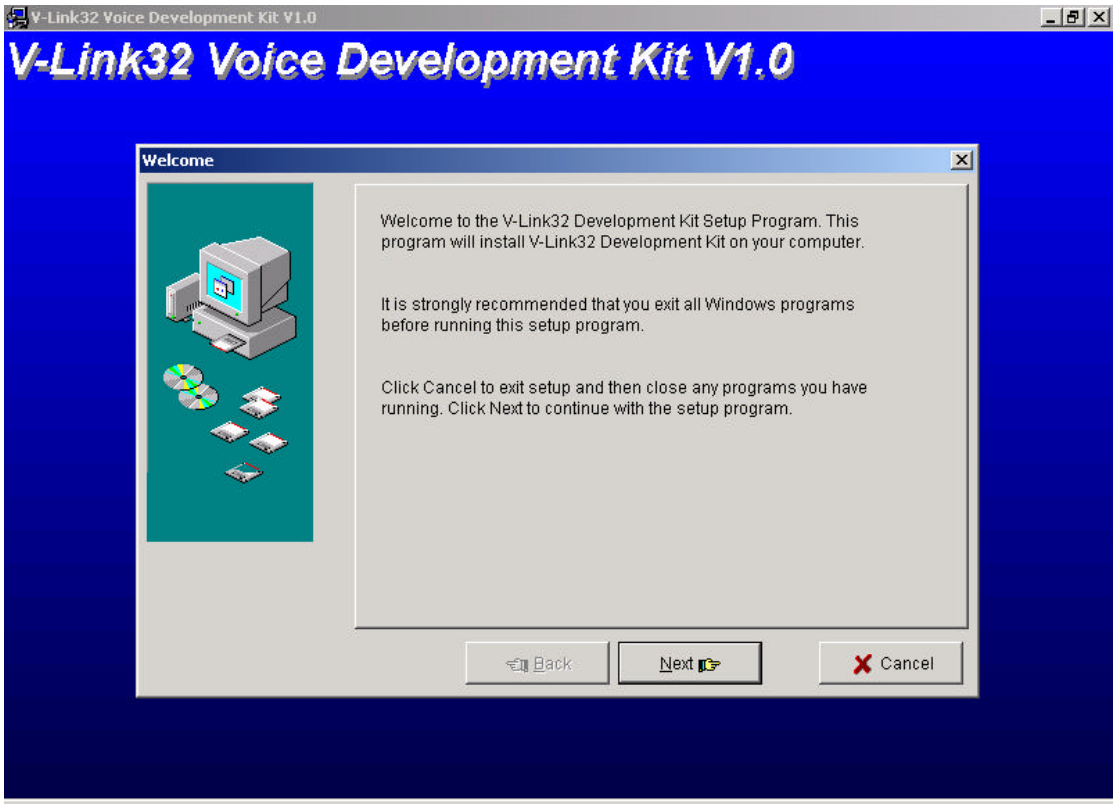
Installation

Install V-Link32 Development Kit

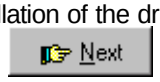
To install the V-Link32 Development Kit on your computer, you must first insert the V-Link32 Voice Setup CDROM into your CDROM drive. If the "Autorun" function does not immediately bring you to the installation screen, please click on Run under the Start menu and type X\INSTALL.EXE (X denoting the letter of your CDROM drive) in the Open field. This will bring you to the following installation screen:

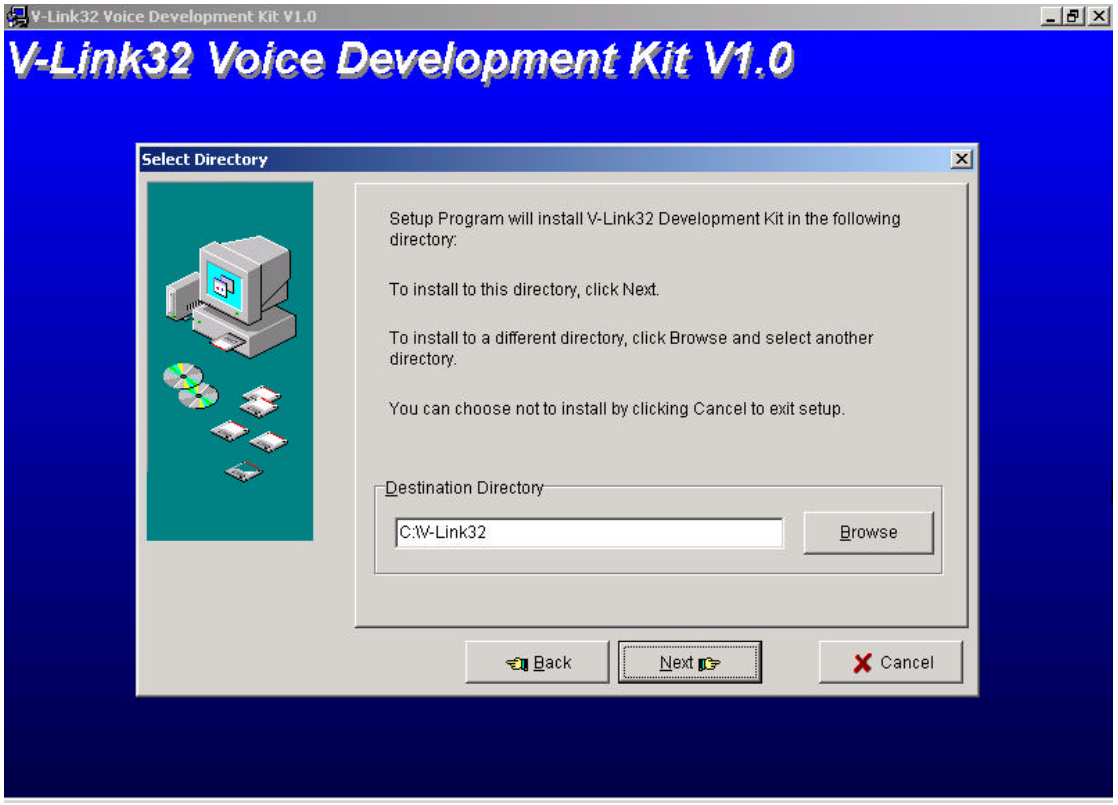

Install
V-Link32 Voice Development Kit V1.0
V-Link32
Voice
Development Kit
Welcome to the V-Link32 Development Kit Setup Program. This program will install V-Link Development Kit on your computer.
It is strongly recommended that you exit all Windows program before running this setup program.
Click Cancel to exit setup and then close any program you have running. Choose the Language you'd like to install and click Next to continue with the setup program.
V-Link32 Drivers (Chinese - Traditional)
V-Link32 Drivers (Chinese - Simplified)
V-Link32 Drivers (English version)
Next
Exit

Please select "V-Link32 Drivers (English version)" and click on th button

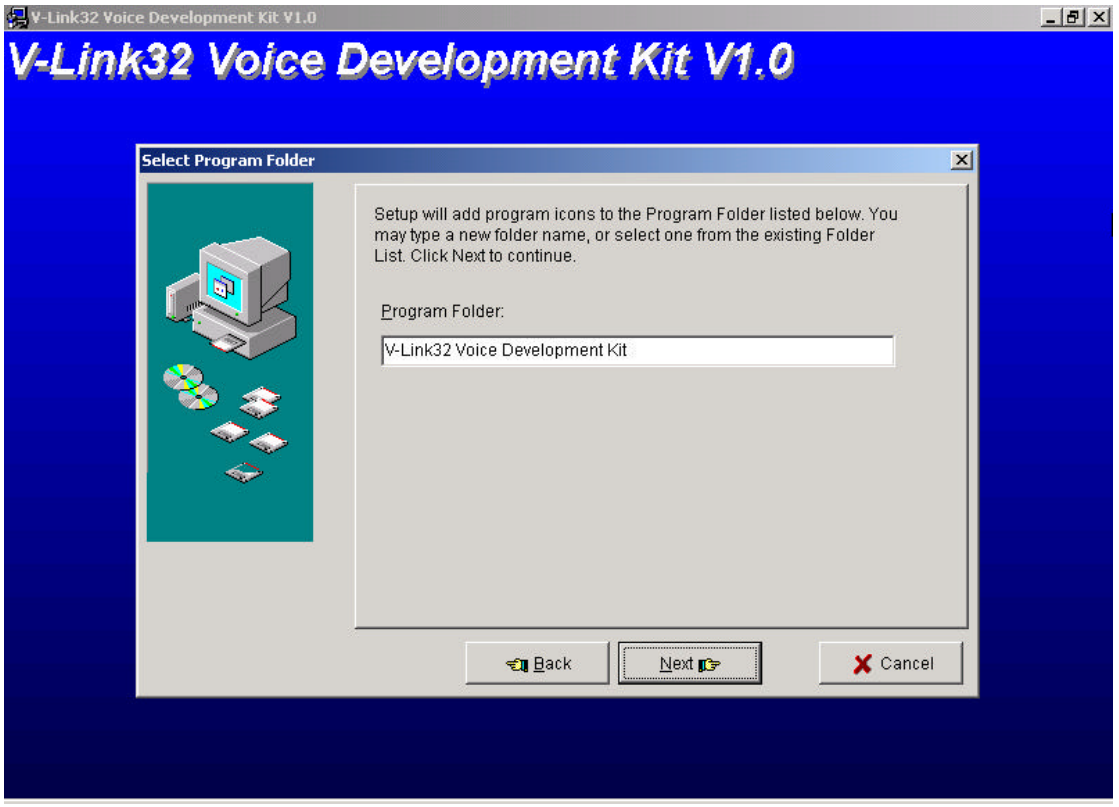

V-Link32 Voice Development Kit V1.0
V-Link32 Voice Development Kit V1.0
Welcome
Welcome to the V-Link32 Development Kit Setup Program. This program will install V-Link32 Development Kit on your computer.
It is strongly recommended that you exit all Windows programs before running this setup program.
Click Cancel to exit setup and then close any programs you have running. Click Next to continue with the setup program.
Back
Next
Cancel

At this screen please exit from all other programs you are currently running to ensure the safe and proper installation of the driver.

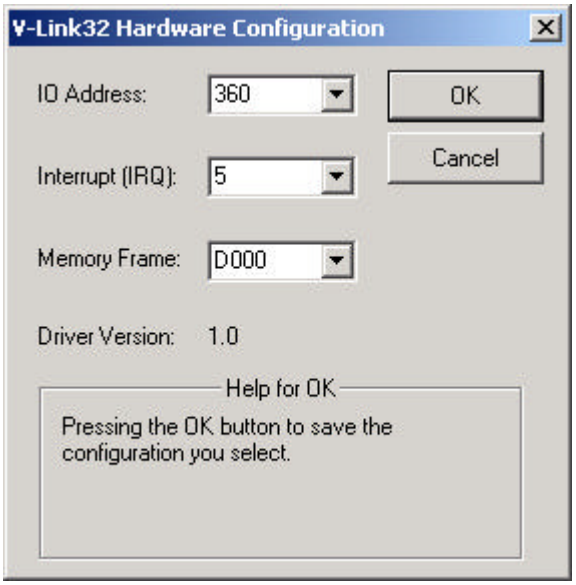
Click on the  button to continue installation.


V-Link32 Voice Development Kit V1.0
V-Link32 Voice Development Kit V1.0
Select Directory
Setup Program will install V-Link32 Development Kit in the following directory:
To install to this directory, click Next.
To install to a different directory, click Browse and select another directory.
You can choose not to install by clicking Cancel to exit setup.
Destination Directory
C:\V-Link32
Browse
Back
Next
Cancel

At this point you can choose the destination directory for your driver program. The default directory is chosen as C:\V-Link32. If you wish to place the drivers somewhere else, indicate the directory in the Destination Directory field, or search for another location by clicking on the Browse button. When you have chosen the directory, click on the Next button to continue.


V-Link32 Voice Development Kit V1.0
V-Link32 Voice Development Kit V1.0
Select Program Folder
Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing Folder List. Click Next to continue.
Program Folder:
V-Link32 Voice Development Kit
Back
Next
Cancel

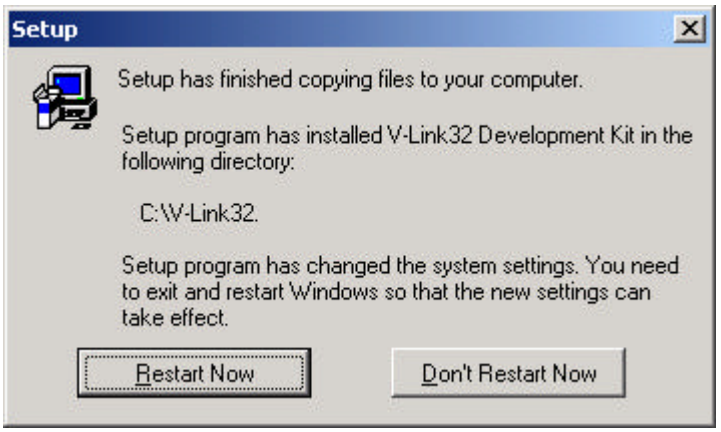
In this screen you can choose which Program Folder you wish to add the V-Link32 Development Kit icons to. Click to move onto the hardware configuration screen.



When this screen appears, you will see the default settings according to the pre-specified jumper settings on your card.
It is advised that you initially use these settings for the installation. If you have already adjusted the jumper settings for your card, please ensure that the settings displayed here match those on your card.

Please move your mouse to the

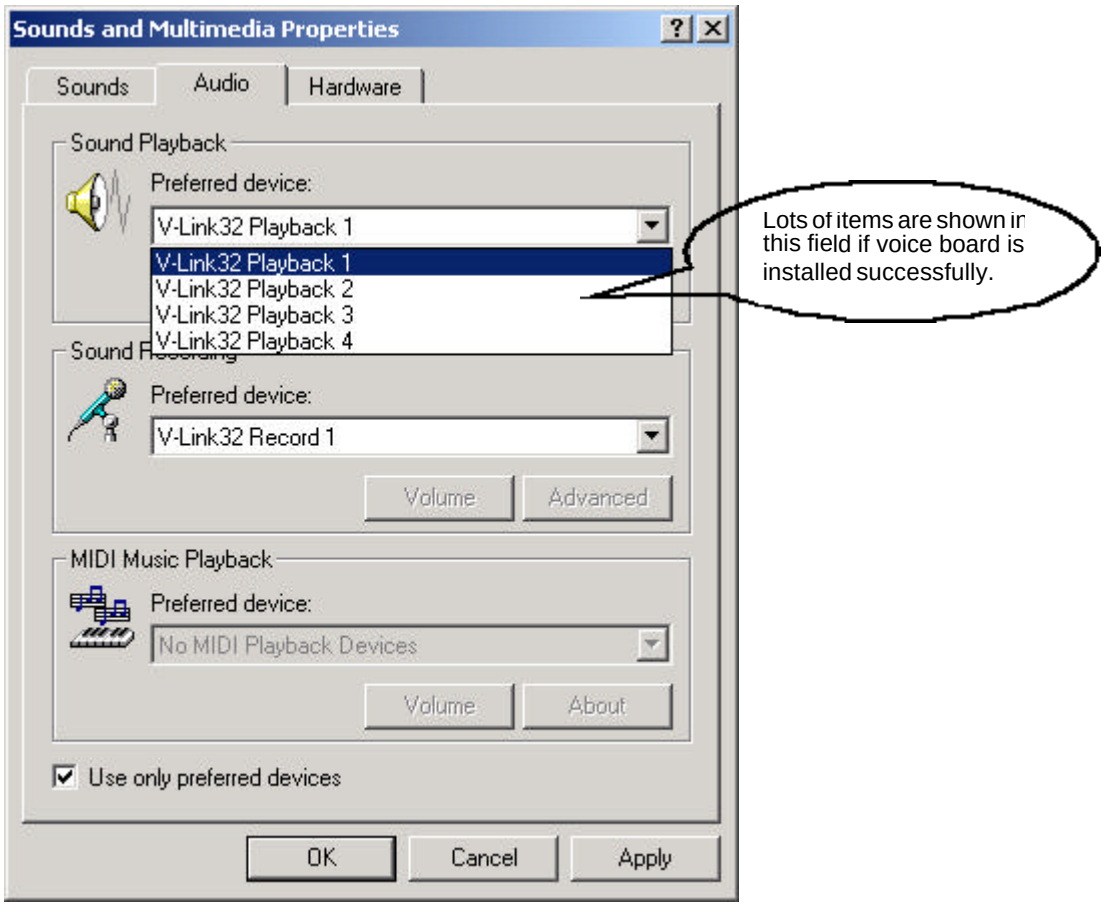
button and click on it to choose the displayed settings and continue with the installation.
You have now completed the installation procedure and can restart Windows so the new settings may take effect


Setup
Setup has finished copying files to your computer.
Setup program has installed V-Link32 Development Kit in the following directory:
C:\V-Link32.
Setup program has changed the system settings. You need to exit and restart Windows so that the new settings can take effect.
Restart Now
Don't Restart Now

To do this, simply move down to the

button and click on it.

After installation is completed and the system has been restarted again, User can run 【Control Panel】 and check the 【Audio】 in 【Multimedia】 to see whether the waveform driver is installed successfully.



If not, please run "Start"->"Programs"->"V-Link32 Development Kit"->"Voice Settings"->"Configure".

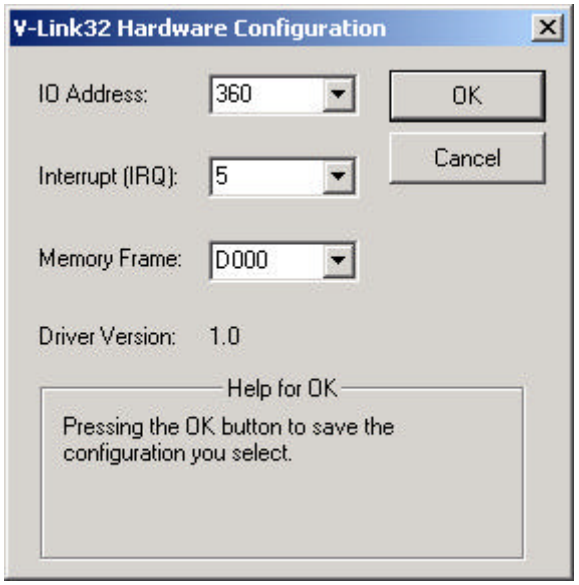
Check if the items of " IO Address" , " Interrupt" and " Shared Memory" match with hardware settigns.

Hardware Settings

The default hardware settings for voice boards are:

IO Port Address: 360
Interrupt (IRQ): 5
Memory Frame: D000

If user want to change the hardware settings, please refer to Hardware Installation Manual for the settings of voice board. After the hardware settings of voice board changed, user also needs to change the hardware configuration.



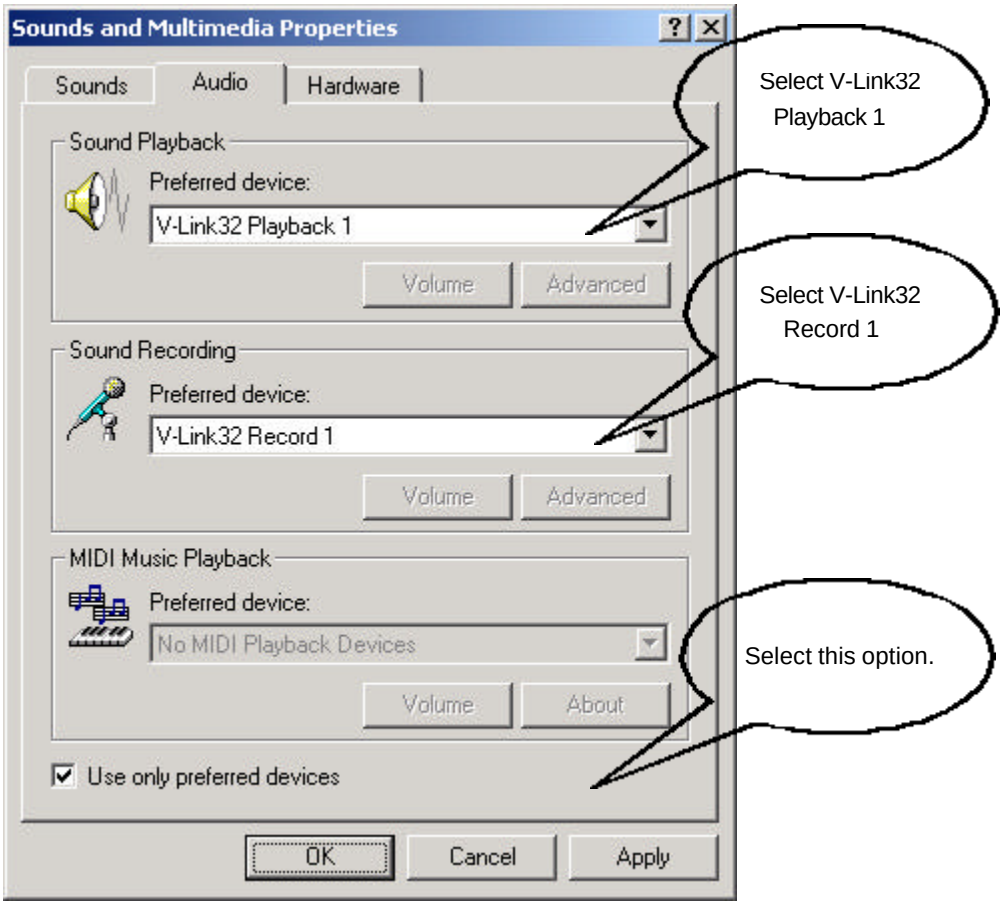
For Windows NT, run SystemService program to change the hardware configuration for voice board.

For Windows 98. use 【System】 of 【Control Panel】 to select the 【Devices】 | 【Media, Audio and Game Controller】, and then find the 【Plus voice/telephony card】 to change the hardware configuration.

Waveform Driver Supported

The voice board supports the waveform driver, so user can use the sound recorder program (sndrec32.exe) of Multimedia to record and play wave files. However the following should also be noted:

- Please use【Control Panel】, select the【Audio】of【Multimedia】to check the Preferred devices of 【Playback】and 【Recording】are the same as below:



- To record with Sound Recorder (sndrec32.exe), it is recommended to use format of PCM with 8,000Hz/ 8-Bit/ Mono, and this will save the time to convert the voice format by the system. The setting can be found in Preferred quality in Audio Properties in Edit of Sound Recorder.

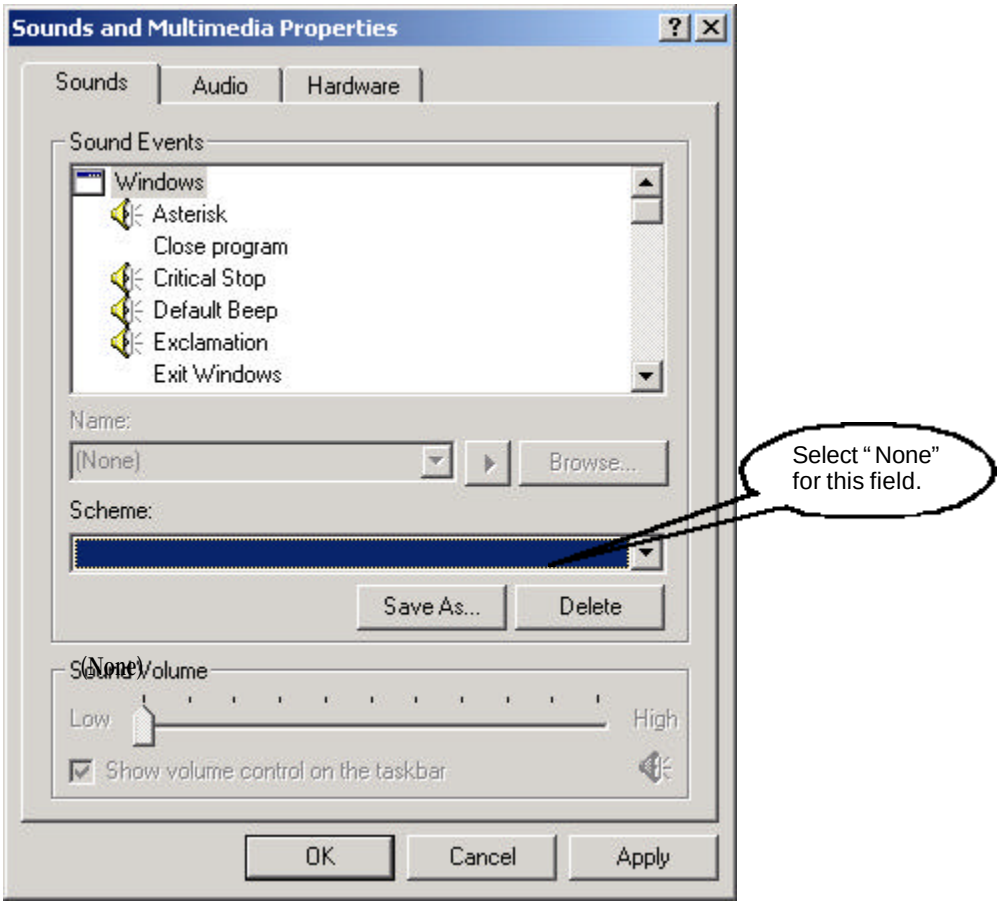
- There have to be no other voice applications running at the same time (voclib32.dll is not running), otherwise you can not record with the phone set and the voice file played will not be heard from the speaker.

In order to avoid the Windows system and the voice applications competing for the same resource of waveform drivers, it is strongly recommended to disa ble the Sounds Events function (as shown below).

If there are other sound cards in your computer and you select it as the player of the Sounds Events, then you can start the system to play Sounds Events Functions.

All in all, you can not set the waveform driver of the voice board as the player of the Sounds Events.

To close the Sounds Events of the system, select 【Sounds】 of【Control Panel】and leave the 【Schemes】blank as below:


Sounds and Multimedia Properties
Sounds
Audio
Hardware
Sound Events
Windows
Asterisk
Close program
Critical Stop
Default Beep
Exclamation
Exit Windows
Name:
(None)
Browse...
Scheme:
Save As...
Delete
Sound Volume
(None)
Low
High
Show volume control on the taskbar
OK
Cancel
Apply
Select “None” for this field.

Introduction

This document describes the Application Programming Interface (API) of the **Plus-series** voice boards. This development kit provides Windows developers with easy-to-use functions and complete interface to control the voice and telephone system.

File List:
The following files are included in this development kit.

<Driver> Directory:

TPLUSDLL.DLL	An installable waveform driver.
TPLUS.SYS	An installable wave/telephony kernel mode driver
DSPCMD.400	A download firmware file.
VOCLIB32.DLL	A Dynamic Link Library for all the voice and telephony functions.
CHNMON32.DLL	A Dynamic Link Library for channel monitor functions.

<Tools> Directory:

DIAG32.EXE	An external configuration program to set the API parameters.
VCAPI32.DLL	A DLL file for DIAG32 program.
DEVICESERVICE.EXE	A program to configure device IO and device removal.
CHNMON32.EXE	A program for monitoring channels.
SIMSIGNAL.EXE	A simulations program for the generation of signals.

<VC++> Directory	For Microsoft Visual C++ programming.
VOCLIB32.H	The include file for ***vocXXX*** functions which is in <VC++\Inc> directory.
CHNMON32.H	The include file for ***monXXX*** functions which is in <VC++\Inc> directory.
ERRORNO.H	The include file for error codes definition which is in <VC++\Inc> directory.
VOCLIB32.LIB	The import library for ***vocXXX*** functions which is in <VC++\Lib> directory. This library is used for Microsoft Visual C++ compiler.
CHNMON32.LIB	The import library for ***monXXX*** functions which is in <VC++\Lib> directory. This library is used for Microsoft Visual C++ compiler.
<BCB> Directory	For Borland C++ Builder programming.
VOCLIB32.H	The include file for ***vocXXX*** functions which is in <BCB\Inc> directory.
CHNMON32.H	The include file for ***monXXX*** functions which is in <BCB\Inc> directory.
ERRORNO.H	The include file for error codes definition which is in <VC++\Inc> directory.
VOCLIB32BC.LIB	The import library for ***vocXXX*** functions which is in <BCB\Lib> directory. This library is used for Borland C++ Builder compiler.
CHNMON32BC.LIB	The import library for ***monXXX*** functions which is in <BCB\Lib> directory. This library is used for Borland C++ Builder compiler.
<VB> Directory	For Microsoft Visual Basic programming.
VOCLIB32.BAS	The module file for ***vocXXX*** functions which is in <VB\Lib> directory. It is used for Microsoft Visual Basic programming.
<Delphi> Directory	For Borland Delphi programming.
VOCLIB32.PAS	The unit file for ***vocXXX*** functions which is in <Delphi\Lib> directory. It is used for Borland Delphi programming.

Function Summary

Device Management Functions:

Function	Description
vocInitDriver()	Initializes the voice driver.
vocEnumChn()	Retrieves the total channels provided by driver.
vocOpenChn()	Opens a free channel.
vocCloseChn()	Closes a opened channel.
vocSetChnParam()	Sets the channel parameters.
vocSetEventCallback()	Redefines the event callback function.
vocGetChnCaps()	Retrieves the capability of the specified channel.
vocGetSerialNo()	Gets the serial number of the specified channel.
vocGetChnID()	Gets the channel number of the specified channel.
vocGetDeviceID()	Gets the waveform device ID of the specified channel.
vocPutSignal()	Simulate the generation of channel' s signal.

Event Management Functions:

Function	Description
vocWaitEvent()	Waiting for a termination event block.
vocGetLastEvent()	Gets the last termination event block.
vocGetLastTerm()	Gets the last termination event code.
vocWaitCST()	Waiting for a CST event block.
vocGetLastCST()	Gets the last CST event block.
vocSetCSTMask()	Sets the CST mask of the specified channel.
vocSetGTDMask()	Sets the GTD mask of the specified channel.
vocGetGTD()	Retrieves the GTD information of the specified channel.
vocQueryGTD()	Retrieves the GTD event for some limited functions.
vocGetLastErr()	Gets the last error code.
VocGetTermDT()	Gets the terminated digit.

I/O Control Functions:

Function	Description
vocWaitRing()	Waiting for the ring signal.
vocSetHook()	Controls the phone line of the specified channel.
vocClearDT()	Clears the channel' s DTMF queue.
vocGetDT()	Collects digits from the channel' s DTMF queue.
vocReadDT()	Get one digit from channel' s DTMF queue.
vocGetChnIO()	Retrieves the I/O status of the specified channel.
vocSetChnIO()	Controls the I/O of the specified channel.
vocFlashHook()	Makes a hook flash.
vocSwitchFax()	Allocates or frees the fax resource of fax daughter board.

Voice Functions:

Function	Description
vocPlayFile()	Plays back the voice file(s).
vocRecordFile()	Records a voice file.
vocPlayTone()	Generates tone.
vocGetCurPos()	Retrieves the current playback or recording position.
vocCutWaveFile()	Truncates the file size.
vocGetVolume()	Retrieves the volume level of the specified channel.
vocSetVolume()	Sets the volume level of the specified channel.
vocStopChn()	Stops the operation of the specified channel.
vocMonitorChn()	Monitors a channel from another channel.

Dial Out Functions:

Function	Description
vocDial()	Dials out a phone number.
vocGetCAR()	Retrieves the call analysis result.

Conference Functions:

Function	Description
vocMakeConference()	Make a conference call.
vocBreakConference()	Terminate a conference call.
vocAddToConference()	Add a channel to conference.
vocDelFromConference()	Delete a channel from conference.
vocGetConfVol()	Get volume level in a conference call.
vocSetConfVol()	Set volume level in a conference call.
vocWaitConfEvent()	Waiting termination event in a conference call.
vocGetConfGTD()	Retrieves the GTD information for a conference call.
vocQueryConfGTD()	Retrieves the GTD event for a conference call.

ADSI Functions:

Function	Description
adsiCAS()	Generates a CAS tone and waits for an ACK tone.
adsiRecvFrame()	Receives a V.23 FSK frame.
adsiSetParam()	Changes channel' s seizure signal and CAS tone settings.
adsiXmitFrame()	Transmits a V.23 FSK frame.

Programming Models

This development kit provides synchronous and asynchronous models for Windows programming. The characteristics of synchronous and asynchronous programming are described below:

Synchronous Programming

Synchronous programming is characterized by functions that block application execution until the function completes. For example, if an application plays back a voice file by calling ***vocPlayFile()*** function, the application will not continue execution until the playing is complete and ***vocPlayFile()*** function has terminated. Since application execution is blocked by a function in the synchronous model, a separate application, thread, or process is needed for each channel.

A sample code of the synchronous model is shown below:

```
main ( ..)
{
HCHN hChn;

// Initialize driver
if (vocInitDriver() != E_OK) {
    /* Process error */
}
// Open a channel with synchronous model and get a channel handle on hChn
if (vocOpenChn(&hChn, ANY_CHN, NULL) != E_OK) {
    /* Process error */
}
if (vocPlayFile(hChn, "voice.wav", 0, 0, 0, DM_SYNC) != E_OK) {
    /* Process error */
}
switch (vocGetLastTerm(hChn)) {
    case EVT_END:
        /* End of playing */
        break;
    case EVT_TERMDT:
        /* Terminated by input digit */
        break;
        :
}
    :
    :
}
```

Asynchronous Programming

In asynchronous programming, multiple channels can be handled in a single thread rather than in separate threads as required in synchronous programming. Handling multiple channels in a single thread results in more efficient use of system resources.

Three types of asynchronous models provides for development include:

- Polling Model
- Function Callback Model
- Window Callback Model

Polling Model (Asynchronous)

In Polling Model, after an asynchronous function is issued, the application polls for and waits for termination events by calling ***vocWaitEvent()*** function. If there is no event, other processing may take place between polls. If any event is available, the event information is returned in the event block.

A sample code of the Polling Model is shown below:

```
main ( ..)
{
HCHN hChn;

// Initialize driver
if (vocInitDriver() != E_OK) {
    /* Process error */
}
// Open a channel with polling model and a channel handle is returned by hChn
if (vocOpenChn(&hChn, ANY_CHN, NULL) != E_OK) {
    /* Process error */
}

if (vocPlayFile(hChn, "voice.wav", 0, 0, 0, DM_ASYNC) != E_OK) {
    /* Process error */
}
    :
// Use vocWaitEvent() function to wait for the completion of vocPlayFile()
vocWaitEvent(hChn, &Event, WT_INFINITE);
    :
or
    :
while (1) {
    if (vocWaitEvent(hChn, &Event, 0) == E_OK) break;
    /* Process other things */
    }
    :
    :
}
```

Function Callback Model (Asynchronous)

In Function Callback Model, after an asynchronous function is issued, the user-defined function procedure will be called when the termination event occurred. The event information can be retrieved by calling ***vocGetLastEvent()*** function.

The declaration of a callback function procedure is described below:

```
void evtCallback(HCHN hChn, DWORD dwMsg, DWORD dwUserData,
                                DWORD dwParam1, DWORD dwParam2);
```

Parameters

hChn

Specifies the channel handle.

dwMsg

Specifies the message value which is defined by the ***dwEventMsg*** parameter of ***vocOpenChn()*** function.

dwUserData

A 32-bit user-instance data which is defined by the ***dwEventCallbackInst*** parameter of ***vocOpenChn()*** function.

dwParam1

Reserved.

dwParam2

Reserved.

A sample code of the Function Callback Model is shown below:

```
main( …)
{
HCHN  hChn;
CBDESC  CB;

// Initialize driver
if (vocInitDriver() != E_OK) {
    /* Process error */
}
// Set the CB
CB.dwEventCallback = (DWORD)evtCallBackProc;
CB.dwEventCallbackInst = NULL ;
CB.dwEventFlag = CB_FUNCTION;
CB.dwEventMsg = NULL;
CB.dwCSTCallback = NULL;
CB.dwCSTCallbackInst = NULL;
CB.dwCSTFlag = NULL;
CB.dwCSTMsg = NULL;
// Open a channel with callback model.
if (vocOpenChn(&hChn, ANY_CHN, &CB) != E_OK) {
    /* Process error */
}
if (vocPlayFile(hChn, " voice.wav ", 0, 0, 0, DM_ASYNC) != E_OK) {
    /* Process error */
}
    :
}


void CALLBACK evtCallbackProc(HCHN hChn, DWORD dw Msg, DWORD dwUserData,
                                          DWORD dwParam1, DWORD dwParam2)
{
EVTBLK  Event;

vocGetLastEvent(hChn, &Event);
switch (Event.wTermFun) {
    case CBT_PLAY:
        // The vocPlayFile() function is completed.
        :
        break;
     case CBT_RECORD:
        // The vocRecordFile() function is completed.
        :
        break;
        :
}
}
```

Window Callback Model (Asynchronous)

In Window Callback Model, after an asynchronous function is issued, the system will send a message to the window handle when the function is complete. The termination event information can be retrieved by calling ***vocGetLastEvent()*** function.

The declaration of callback function is described below:

void WndProc(HWND hWnd, UNIT Message, UNIT wParam, LONG lParam);

Parameters

hWnd

Specifies the window handle.

Message

Specifies the message value which defined by the ***dwEventMsg*** parameter of ***vocOpenChn()*** function.

wParam
This field contains the channel handle.

lParam
Reserved.

A sample code of the Window Callback Model is shown below:

```
#define WM_TERMNOTIFY      (WM_USER+1)

main( …)
{
HWND hWnd;
HCHN hChn;
CBDESC  CB;
    :
// Create window
hWnd = CreateWindow(szWndClass,
                "Sample Application",
                CW_USEDEFAULT, CW_USEDEFAULT,
                CW_USEDEFAULT, CW_USEDEFAULT,
                NULL, NULL, hInstance, NULL};

// Initialize driver
if (vocInitDriver() != E_OK) {
    /* Process error */
}
// Set the CB
CB.dwEventCallback = (DWORD)hWnd;
CB.dwEventCallbackInst = NULL;
CB.dwEventFlag = CB_WINDOW;
CB.dwEventMsg = WM_TERMNOTIFY;
CB.dwCSTCallback = NULL;
CB.dwCSTCallbackInst = NULL;
CB.dwCSTFlag = NULL;
CB.dwCSTMsg = NULL;
// Open a channel with callback model.
if (vocOpenChn(&hChn, ANY_CHN, &CB) != E_OK) {
    /* Process error */
}
    :
if (vocPlayFile(hChn, "voice.wav", 0, 0, 0, DM_ASYNC) != E_OK) {
    /* Process error */
}
    :
}


long PASCAL WndProc(HWND hWnd, UINT Message, UINT w Param, LONG lParam)
{
HCHN hChn;
EVENTBLK  Event;

switch (Message) {
    case WM_CREATE:
        :
        break;
    case WM_COMMAND:
        :
        break;
    case WM_TERMNOTIFY:
        hChn = (HCHN) wParam;
        vocGetLastEvent(hChn, &Event);
        switch (Event.wTermFun) {
            case CBT_PLAY:
            // The vocPlayFile() function is completed.
            :
            break;
             case CBT_RECORD:
```

```
            // The vocRecordFile() function is completed.
            :
            break;
            :
        }
        break;
        :
}
}
```

Event Management

This development kit provides two event types: ***Termination Event*** and ***CST Event***.

Termination Event

Every asynchronous function, such as ***vocPlayFile()*** or ***vocRecordFIle()***, will generate a termination event to indicate the function is complete. The termination events can be retrieved by calling ***vocWaitEvent()***, ***vocGetLastEvent()*** or ***vocGetLastTerm()*** function.

The defines of termination events are listed below:

Termination Event	Code	Description
EVT_END	x0000	Function is terminated successful.
EVT_ERR	XFFFF	Function is terminated due to an error. Call ***vocGetLastErr()*** to retrieve the reason of error.
EVT_GTD	x1001	Function is terminated due to a GTD tone detected. Call ***vocGetGTD()*** to retrieve the reason of GTD detection.
EVT_MAXDTMF	x1002	The maximum number of digits has received.
EVT_IDDTIME	x1003	Inter-digit delay time elapsed.
EVT_MAXTIME	x1004	Maximum function time elapsed.
EVT_STOP	x1005	Stopped by ***vocStopChn()*** function.
EVT_TERMDT	X1006	A termination digit terminates function. Call ***vocGetTermDT()*** to retrieve the terminated digit.

CST Event

The CST events are generated when the channel status transition is changed. The CST events can be retrieved by calling ***vocWaitCST()*** or ***vocGetLastCST()*** function.

The defines of CST events are listed below:

CST Event	Meaning
CST_RING	A ring signal is detected. The ***wCSTData*** contains the number of rings detected.
CST_DIGIT	A DTMF digit is detected. The ***wCSTData*** specifies the ASCII digit. (0 – 9, *, #, A – D)
CST_SILON	Silence duration is starting. The ***wCSTData*** specifies the interval for non-silence time. (in 10 ms)
CST_SILOFF	Non-silence duration is starting. The ***wCSTData*** specifies the interval for silence time. (in 10 ms)
CST_ONHOOK	On-hook state is detected.
CST_OFFHOOK	Off-hook state is detected.
CST_LCREV	Loop current reversal is detected. The ***wCSTData*** contains the reverse state. (LC_NORM2REV or LC_REV2NORM)
CST_LCDROP	Loop current drop is detected.
CST_LCON	Loop current is now on . The ***wCSTData*** specifies the interval time for loop current off. (in 10 ms)
CST_LCOFF	Loop current is now off. The ***wCSTData*** specifies the interval time for loop current on. (in 10 ms)
CST_RINGON	The leading edge of ring signal is detected. The ***wCSTData*** specifies the interval time for ring off. (in 10 ms)
CST_RINGOFF	The falling edge of ring signal is detected. The ***wCSTData*** specifies the

	interval time for ring on. (in 10 ms)
CST_TONEON	A user-defined tone is detected. The ***wCSTData*** specifies the user-defined Tone ID.

The CST events are very useful for some applications to handle the channel status, but it is not necessary for all applications. In general, applications may ignore the CST events, if they don't need to monitor the channel status.

There are three programming models to retrieve the CST event:

- Polling Model
- Function Callback Model
- Window Callback Model

Polling Model

In Polling Model, application polls for CST events by calling ***vocWaitCST()*** function.

An example code of the Polling Model is shown below:

```
main ( ..)
{
HCHN  hChn;
CSTBLK  CST;

// Initialize driver
if (vocInitDriver() != E_OK) {
    /* Process error */
}
// Open a channel with polling model and a channel handle is returned by hChn
if (vocOpenChn(&hChn, ANY_CHN, NULL) != E_OK) {
    /* Process error */
}
    :
// Use vocWaitCST() function to retrieve CST events
vocWaitCST(hChn, &CST, WT_INFINITE);
    :
or
    :
while (1) {
    if (vocWaitCST(hChn, &CST, 100) == E_OK) break;
    /* Process other things */
    }
    :
}
```

Function Callback Model

In Function Callback Model, after the channel status transition is changed, the user-defined function procedure will be called when a CST event occurred. The event information can be retrieved by calling ***vocGetLastCST()*** function.

The declaration of a callback function procedure is described below:

```
void cstCallback(HCHN hChn, DWORD dwMsg, DWORD dwUserData,
                                DWORD dwParam1, DWORD dwParam2);
```

Parameters

hChn

Specifies the channel handle.

dwMsg

Specifies the message value which is defined by the ***dwCSTMsg*** parameter of ***vocOpenChn()*** function.

dwUserData

A 32-bit user-instance data which is defined by the ***dwCSTCallbackInst*** parameter of ***vocOpenChn()*** function.

dwParam1

Reserved.

dwParam2

Reserved.

A sample code of the Function Callback Model is shown below:

```
main( …)
{
HCHN  hChn;
CBDESC  CB;

// Initialize driver
if (vocInitDriver() != E_OK) {
    /* Process error */
}

// Set the CB
CB.dwEventCallback = NULL;
CB.dwEventCallbackInst = NULL ;
CB.dwEventFlag = NULL;
CB.dwEventMsg = NULL;
CB.dwCSTCallback = (DWORD)cstCallBackProc;
CB.dwCSTCallbackInst = NULL;
CB.dwCSTFlag = CB_FUNCTION;
CB.dwCSTMsg = NULL;

// Open a channel with callback model.
if (vocOpenChn(&hChn, ANY_CHN, &CB) != E_OK) {
    /* Process error */
}
    :
}


void CALLBACK cstCallbackProc(HCHN hChn, DWORD dw Msg, DWORD dwUserData,
                                                    DWORD dwParam1, DWORD dwParam2)
{
CSTBLK  CST;

vocGetLastCST(hChn, &CST);
switch (CST.wStatus) {
    case CST_RING:
        :
        break;
    case CST_DIGIT:
        :
        break;
        :
}
}
```

Window Callback Model

In Window Callback Model, after the channel status transition is changed, the system will send a message to the window handle when a CST event occurred. The event information can be retrieved by calling ***vocGetLastCST()*** function.

The declaration of callback function is described below:

void WndProc(HWND hWnd, UNIT Message, UNIT wParam, LONG lParam);

Parameters

hWnd

Specifies the window handle.

Message

Specifies the message value which defined by the ***dwCSTMsg*** parameter of ***vocOpenChn()*** function.

wParam

This field contains the channel handle.

lParam

Reserved.

An sample code of the Window Callback Model is shown below:

```
#define WM_CSTNOTIFY        (WM_USER+2)

main( …)
{
HWND hWnd;
HCHN hChn;
CBDESC CB;

    :
// Create window
hWnd = CreateWindow(szWndClass,
                "Sample Application",
                CW_USEDEFAULT, CW_USEDEFAULT,
                CW_USEDEFAULT, CW_USEDEFAULT,
                NULL, NULL, hInstance, NULL};

// Initialize driver
if (vocInitDriver() != E_OK) {
    /* Process error */
}
// Set the CB
CB.dwEventCallback = NULL;
CB.dwEventCallbackInst = NULL ;
CB.dwEventFlag = NULL;
CB.dwEventMsg = NULL;
CB.dwCSTCallback = (DWORD)hWnd;
CB.dwCSTCallbackInst = NULL;
CB.dwCSTFlag = CB_WINDOW;
CB.dwCSTMsg = WM_CSTNOTIFY;
// Open a channel with callback model.
if (vocOpenChn(&hChn, ANY_CHN, &CB) != E_OK) {
    /* Process error */
}
    :
}


long PASCAL WndProc(HWND hWnd, UNIT Message, UINT w Param, LONG lParam)
{
HCHN  hChn;
CSTBLK  CST;

switch (Message) {
    case WM_CREATE:
        :
        break;
    case WM_COMMAND:
        :
```

```
        break;
    case WM_CSTNOTIFY:
        // Process CST event
        hChn = (HCHN) wParam;
         vocGetLastCST(hChn, &CST);
        :
        break;
        :
}
}
```

Function Return Codes

This section lists the return codes that returned from the function calls.

Value	Code	Description
E_OK	X0000	Function is successful.
E_ERR	XFFFF	Function error.
E_CHNERR	X1001	Invalid channel handle.
E_NOMEM	X1002	Insufficient memory.
E_BUSY	X1003	Channel is in use.
E_NOFILE	X1004	Voice file is not found.
E_READERR	X1005	Fail to read voice file.
E_WRTERR	X1006	Fail to write voice file.
E_FMTERR	X1007	Unknown media format.
E_CREATEERR	X1008	Fail to create voice file.
E_DRVERR	X1009	Fail to initialize the device driver.
E_BADPARAM	X100A	Invalid input parameters.
E_TIMEOUT	X100B	The time-out interval lapses.
E_ALLOCATED	X100C	The specified channel has been allocated.
E_CONFERR	X100D	Invalid conference handle.
E_CHNINUSE	X100E	Channel was in conference already.
E_TOOMANYCHN	X100F	Too many channels in one conference call.
E_CONFFULL	X1010	Out of conference handle.
E_SYSERR	X1011	Fail to synchronize.
E_NOCHN	X1012	No channel available to be used.
E_IOERR	X1013	Fail to communicate with driver.
E_RECINUSE	X1014	The channel has been already monitored by another channel.
E_FUNERR	X1015	No conference call function supported.

Function Description

This paragraph contains an alphabetical list of the API functions. The documentation for each function contains a line illustrating correct syntax, a statement about the function's purpose, a description of its input parameters, and a description of its return value. The documentation for some functions contains additional, important information that an application developer needs in order to use the function.

adsiCAS

This function generates a CPE Alert Signal (CAS) to the remote device and waits for an acknowledge signal (ACK Tone).

Syntax

```
WORD adsiCAS (
    HCHN hChn,
    DWORD dwMaxTime,
    WORD wAckTones,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

dwMaxTime

Specifies the time-out interval, in milliseconds. If this parameter is WT_INFINITE, the function's time-out interval never elapses. If the interval elapses and no acknowledge signal is detected, an EVT_MAXTIME event will be generated.

wAckTones

Specifies the acknowledge signals (ACK tones) to receive, in which the reception of any DTMF tone will terminate this function. This parameter can be a combination of the following values:

DT_0 : Digit 0.	**DT_6 :** Digit 6.	**DT_A :** Digit A.
DT_1 : Digit 1.	**DT_7 :** Digit 7.	**DT_B :** Digit B.
DT_2 : Digit 2.	**DT_8 :** Digit 8.	**DT_C :** Digit C.
DT_3 : Digit 3.	**DT_9 :** Digit 9.	**DT_D :** Digit D.
DT_4 : Digit 4.	**DT_S :** Digit *.	**DT_ALL :** For all digits.
DT_5 : Digit 5.	**DT_P :** Digit #.	

wMode

Specifies the running mode and options. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one or more:

Value	Meaning
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD events.
DM_NOCAS	Don't generate a CAS tone, just waiting for ACK tone.
DM_NOCLR	Don't clear the channel's DTMF queue before waiting ACK tone.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	End of CAS tone playing and an ACK tone is detected successfully. If multiple digits is defined for ***wAckTones***, call ***vocGetTermDT()*** to get the terminated digit (i.e. ACK tone).
EVT_MAXTIME	Waiting time is reached and no ACK tone is detected.
EVT_GTD	GTD Tone detected. Call **vocGetGTD()** function to get detail information.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.

Remarks

This function is used to communicate with the remote devices. The more detail information about ADSI protocol, please refer to the ADSI section in Application Notes.

The default Call Alert Signal (CAS Tone) is a dual-frequency tone (2130 Hz and 2750 Hz) which can be programming by calling the ***adsiSetParam()*** function. The Acknowledge signal (ACK Tone) is specified by the ***wAckTones***, and it is always one of the DTMF digits.

When the ***adsiCAS()*** function is executing, all the input digits will be read out from the channel's DTMF queue including the terminated digit (ACK tone). In order to prevent from getting an unexpected ACK tone, the ***adsiCAS()*** function will clear the DTMF queue before waiting for an ACK tone. If applications allow an ACK tone occurred before the ***adsiCAS()*** function call, specifies the ***wMode*** with DM_NOCLR option.

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN hChn;

// Generates a CAS and waits for ACK tone (DTMF digit "A")
If (adsiCAS(hChn, 3000, DT_A, DM_SYNC) != E_OK) {
    // Process error.
    }
switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_MAXTIME:
        // Time-out, no ACK tone found.
        goto Error;
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }
```

Example 2: Using function in asynchronous model.

```
HCHN hChn;
EVTBLK Event;

// Generates a CAS and waits for ACK tone (DTMF digit " A")
If (adsiCAS(hChn, 3000, DT_A, DM_ASYNC) != E_OK) {
    // Process error.
    }

// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);

switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_MAXTIME:
        // Time-out, no ACK tone found.
        goto Error;
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }
```

adsiCheckSum

This function computes the check sum byte for a ADSI message.

Syntax

BYTE adsiCAS (
LPSTR pFrame,
WORD wLength
);

Parameters

PFrame

Points to a buffer where contains the ADSI message.

wLength

Specifies the total length of buffer to compute the check sum.

Return Values

Returns a check sum byte for input frame message.

Remarks

This function is used to compute the check sum byte for a ADSI message. The check sum byte is a two's complement sum of all bytes starting from the ADSI message up to the end of the message block.

Example

```
#define DATA_LEN      20

typedef struct tagADSIFrame {
    BYTE bType;
    BYTE bLength;
    BYTE bData[DATA_LEN];
    BYTE bCS;
    } ADSIFRAME;

ADSIFRAME Frame;
WORD wMsgSize;

Frame.bType = 0xAA;
Frame.bLength = DATA_LEN;
// Fill data into Frame.bData;

Frame.bCS = adsiCheckSum(&Frame, sizeof(ADSIFRAME) -1); // -1 is used to exclude the CheckSum byte.
// Frame is ready to transmit.
AdsiXmitFrame(hChn, &Frame, sizeof(ADSIFRAME), DM_SYNC);
    :
    :



// After a frame received
adsiRecvFrame(hChn, &Frame, sizeof(ADSIFRAME), &wMsgSize, 5000, DM_RAW|DM_SYNC);
    :
    :
if (adsiCheckSum(&Frame, sizeof(ADSIFRAME)-1) != Frame.bCS) // -1 is used to exclude the CheckSum byte.
    {
    // Error on Check Sum
    }
    :
    :
```

adsiRecvFrame

This function receives a V.23 FSK frame in raw data format.

Syntax

```
WORD adsiRecvFrame (
    HCHN hChn,
    LPSTR pBuf,
    WORD wBufSize,
    LPWORD pRecvSize,
    DWORD dwMaxTime,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

pBuf

Points to a buffer to receive the message data. The length of input buffer must be greater than the ***wBufSize*** value to prevent from an unexpected error. The received data only contains the message data exculding the seizure and mark signals. Application should verify the Check Sum field of message data to confirm the received data.

wBufSize

Specifies the maximum number of bytes to receive the message data. The valid value is from 1 to 256 bytes.

pRecvSize

Points to a word buffer, which will be filled with the length of received message including the check sum byte.

dwMaxTime

Specifies the time-out interval, in milliseconds. If this parameter is WT_INFINITE, the function's time-out interval never elapses. If the interval elapses and no frame is received, an EVT_MAXTIME event will be generated.

wMode

Specifies the running mode and options. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one or more:

Value	Meaning
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD events.
DM_RAW	Don't check the CheckSum byte which is located at the last byte of received message.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	A frame is detected sucessfully.
EVT_ERR	A frame is detected but the check sum is error. The input buffer (***pBuf***) will still contains the received message and the ***pRecvSize*** also be filled with the length of the received message. Driver will not generate this event if **DM_RAW** option is specified.
EVT_MAXTIME	Waiting time is reached and no frame is detected.
EVT_GTD	GTD Tone detected. Call **vocGetGTD()** function to get detail information.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.

Remarks

The definition of received message is depending on ADSI protocol. In general, the received message consists of Message Type, Message Length, Message Data and Check Sum byte for ADSI application:

Message Type	Message Length	Message Data	Check Sum

To ensure the correction of received data, application can call ***adsiCheckSum()*** function to verify the Check Sum byte which is usually located at the last byte of received message.

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN hChn;
Char Buf[100];
WORD wMsgSize;

// Receive a V.23 FSK frame. Function will be stopped by either 10 seconds reaches, or any frame detected.
if (adsiRecvFrame(hChn, Buf, sizeof(Buf), &wMsgSize, 10000, DM_SYNC) != E_OK) {
    // Process error
}

switch(vocGetLastTerm()) {
    case EVT_END:
        // A frame detected.
        break;
    case EVT_ERR:
        // A frame detected, but check sum is error.
        break;
    case EVT_MAXTIME:
```

```
        //Time out and no frame detected.
        goto error;
        break;
    case EVT_GTD:
        //Caller have alreay disconnected the line or some signals is detected.
        goto HangUp;
        break;
    default:
        goto error;
        break;
    }
```

Example 2: Using function in asynchronous model.

```
HCHN hChn;
Char Buf[100];
EVTBLK Event;
WORD wMsgSize;

// Receive a V.23 FSK frame. Function will be stopped by either 10 seconds reaches, or any frame detected.
if (adsiRecvFrame(hChn, Buf, sizeof(Buf), &wMsgSize, 10000, DM _ASYNC) != E_OK) {
    // Process error
}

// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);

switch(Event.wEvent) {
    case EVT_END:
        // A frame detected.
        break;
    case EVT_ERR:
        // A frame detected, but check sum is error.
        break;
    case EVT_MAXTIME:
        //Time out and no frame detected.
        goto error;
        break;
    case EVT_GTD:
        //Caller have alreay disconnected the line or some signals is detected.
        goto HangUp;
        break;
    default:
        goto error;
        break;
    }
```

adsiSetParam

This function sets the channel' s ADSI parameters.

Syntax

```
WORD adsiSetParam (
    HCHN hChn,
    WORD wParam,
    DWORD dwData
    );
```

Parameters

hChn

Identifies the channel handle.

wParam

Specifies the ADSI parameter to change. This parameter can be one of the following values:

Values	Meaning
ADSI_RECV_SEIZURE	Set the alternating seizure bits for ADSI frame receiving.
ADSI_XMIT_SEIZURE	Set the alternating seizure bits for ADSI frame transmission.
ADSI_CAS_DURATION	Set the CAS tone duration.
ADSI_CAS_FREQ1	Set the 1st frequency of CAS tone.
ADSI_CAS_FREQ2	Set the 2nd frequency of CAS tone.
ADSI_CAS_AMP1	Set the amplitude for the 1st frequency.
ADSI_CAS_AMP2	Set the amplitude for the 2nd frequency.

dwData

Contains the data associated with ***wParam***. This field has a different definition for each channel parameter:

wParam	dwData
ADSI_RECV_SEIZURE	Specifies the at least number of alternating seizure bits for ADSI frame detection. The default value is 50. (i.e. 50 alternating 0 and 1 bits to indicate the beginning of a ADSI frame.) The seizure count must be a multiple of ten (i.e. 10, 20, 30 or 300). To prevent from receiving the seizure signal, set this value to zero.
ADSI_XMIT_SEIZURE	Specifies the at least number of alternating seizure bits for ADSI frame transmission. The default value is 50. (i.e. 50 alternating 0 and 1 bits will automatically be added preceding transmit a ADSI frame.) The seizure count must be a multiple of ten (i.e. 10, 20, 30 or 300). To prevent from generating the seizure signal, set this value to zero.
ADSI_CAS_DURATION	Specifies the duration of CAS tone and time unit is 10 ms. The default value is 10. (i.e. 100ms tone duration)
ADSI_CAS_FREQ1	Specifies the 1st frequency of CAS tone. The valid value is from 200 - 4000 and unit is Hz. The default value is 2130 Hz.
ADSI_CAS_FREQ2	Specifies the 2nd frequency of CAS tone. If the CAS tone is a dual-frequency, specifies the 2nd frequency for this field. If the CAS tone is a single frequency, fill this field with zero. The valid value is from 200 - 4000 and unit is Hz. The default value is 2750 Hz.
ADSI_CAS_AMP1	Specifies the amplitude for the 1st frequency. The valid value is from -40 to 0 and unit is dB. The default value is –9 dB.

ADSI_CAS_AMP2	Specifies the amplitude for the 2nd frequency. The valid value is from -40 to 0 and unit is dB. The default value is -11dB.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_ERR	Requested function is not supported.

Example:

```
HCHN  hChn;

// Set the IDD time to 2 seconds
if (adsiSetParam(hChn, ADSI_RECV_SEIZURE, 50) != E_OK) {
    // Process error,
}

// Set the frequeny of CAS tone
adsiSetParam(hChn, ADSI_CAS_FREQ1, 2300);
adsiSetParam(hChn, ADSI_CAS_FREQ2, 2850);
```

adsiXmitFrame

This function transmits a message in V.23 FSK frame.

Syntax

```
WORD adsiXmitFrame (
    HCHN hChn,
    LPSTR pBuf,
    WORD wBufSize,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

pBuf

Points to a buffer where contains the message to transmit. When the frame is transmitting, driver will automatically add the seizure and mark signals to the beginning of FSK frame. The contents of the message prepared by the application should include message type, message length, message data and check sum byte for ADSI protocol.

wBufSize

Specifies the number of bytes to transmit (i.e. the length of message including message type, message length, message data and check sum byte).

wMode

Specifies the running mode and options. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one or more:

Value	Meaning
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD events.
DM_RAW	Don' t add the checksum byte into ***pBuf***.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.
E_NOMEM	Not enough memory to allocate.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	The frame data is transmitted.
EVT_ERR	I/O Device error. Calls ***vocGetLastErr()*** to get detail error code.
EVT_GTD	GTD Tone detected. Call **vocGetGTD()** function to get detail information.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.

Remarks

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN hChn;
char Frame[100];

// Transmit a frame.
if (adsiXmitFrame(hChn, Frame, sizeof(Frame), DM_SYNC) != E_OK) {
    //Process error;
    }
switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }
```

Example 2: Using function in asynchronous model.

```
HCHN hChn;
EVTBLK Event;
char Frame[100];

// Transmit a frame.
if (adsiXmitFrame(hChn, Frame, sizeof(Frame), DM_ASYNC) != E_OK) {
    //Process error;
    }

// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);

switch (Event.wEvent) {
    case EVT_END:
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }
```

anaGetXmitSlot

This function returns SCbus time slot number of analog transmit channel. It returns the SCbus time slot information contained in a TSINFO structure that includes the number of the SCbus time slot connected to the voice transmit channel on a PLUS-4LVSC board.

Syntax

```
WORD vocGetXmitSlot (
   HCHN hChn,
   LPTSINFO lpTSInfo
   );
```

Parameters

hChn

Identifies the channel handle.

lpTSInfo

Points to a TSINFO structure to receive timeslot number in ***pTsArray***. The TSINFO structure has the following form:

```
typedef struct tagTSINFO {
  DWORD   nTsCnt;
  long  *pTsArray;
} TSINFO;
```

The detail description of this structure is listed below:

nTsCnt

This parameter must be initialized with the number of SCbus time slots, typically 1.

pTsArray

This parameter must be initialized with a pointer to a valid array. Upon return from the function, the array will contain the number (between 0 and 1023) of the SCbus time slot on which the analog channel transmits.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_BADPARAM	Invalid input parameter. The ***nTsCnt*** parameter is not equal to 1.
E_BADTYPE	Invalid channel type (voice, analog, fax ..etc)
E_FUNERR	Function is not supported in current bus configuration.

Remarks

A analog channel on a PLUS-4LVSC board can transmit on only one SCbus time slot.

Example

```
HCHN  hChn;
TSINFO TsInfo;

TsInfo.nTsCnt = 1;   //the number of timeslot to get is one
if (anaGetXmitSlot(hChn, &TsInfo) == E_OK) {
    printf(" \n\rThe transmit time slot for analog=%d" , TsInfo.pTsArray[0]);
}
```

anaListen

This function connects analog listen channel to SCbus time slot. This function uses the information stored in the TSINFO structure to connect the receive analog (listen) channel on a PLUS-4LVSC board to an SCbus time slot.

Syntax

```
WORD anaListen (
   HCHN hChn,
   LPTSINFO lpTSInfo
   );
```

Parameters

hChn

Identifies the channel handle.

lpTSInfo

Points to a TSINFO structure to specify the time slot information.. The TSINFO structure has the following form:

```
typedef  struct  tagTSINFO {
  DWORD   nTsCnt;
  long  *pTsArray;
} TSINFO;
```

The detail description of this structure is listed below:

nTsCnt

This parameter must be set to 1.

pTsArray

This parameter must be initialized with a pointer to a valid array. The first element of this array must contain a valid SCbus time slot number (between 0 and 1023) which was obtained by issuing an ***vocGetXmitSlot()*** function. Upon return from this function, the analog receive channel will be connected to the SCbus time slot.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_BADPARAM	Invalid input parameter. The ***nTsCnt*** parameter is not equal to 1.
E_BADTYPE	Invalid channel type (voice, analog, fax ..etc)
E_FUNERR	Function is not supported in current bus configuration.

Remarks

This function sets up a half-duplex connection. For a full-duplex connection, the receive (listen) channel of the other device must be connected to the analog transmit channel.

Although multiple analog channels may listen (be connected) to the same SCbus time slot, the receive of a analog channel can connect to only one SCbus time slot.

Calling the ***anaListen()*** function to connect to a different SCbus time slot will automatically break an existing connection. Thus, when changing connections, you need not call the ***vocUnlisten()*** function.

Example

```
HCHN hChn;
TSINFO TsInfo;
TsInfo.nTsCnt = 1;
TsInfo.pTsArray[0] = 512; // if the timeslot going to listen is 512
```

anaListen( )

```
if (anaListen(hChn, &TsInfo) == E_OK) {
    printf(“ \n\rAnalog channel is ready to listen time slot 512!” );
}
```

anaUnlisten

This function disconnects analog receive channel from SCbus. This function disconnects the analog receive (listen) channel on a PLUS-4LVSC board from the SCbus.

Syntax

```
WORD anaUnlisten (
    HCHN hChn
    );
```

Parameters

hChn

Identifies the channel handle.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_FUNERR	Function is not supported in current bus configuration.

Remarks

Calling the ***anaListen()*** function to connect to a different SCbus time slot will automatically break an existing connection. Thus, when changing connections, you need not call the ***vocUnlisten()*** function.

Example

```
HCHN hChn;
if (anaUnlisten(hChn) == E_OK) {
    printf(" \n\rAnalog channel is no longer listening!" );
}
```

vocAddToConference

This function adds a channel to the specified conference call.

Syntax

```
WORD vocAddToConference(
    HCONF hConf,
    HCHN hChn
    );
```

Parameters

hConf

Identifies the conference handle.

hChn

Identifies the channel handle to add.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_CONFERR	Invalid conference handle.
E_CHNINUSE	One of the input channels is in use for other conference call.
E_TOOMANYCHN	Too many channels in a conference handle.
E_FUNERR	No conference call function supported.
E_NOMEM	Insufficient memory.

Example

```
HCONF hConf;
HCHN  hChn;

if (vocAddToConference(hConf, hChn) != E_OK) {
    /* Process error */
}
```

vocBreakConference

This function terminates a conference call.

Syntax

```
WORD vocBreakConference(
    HCONF hConf
    );
```

Parameters

hConf
Identifies the conference handle.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CONFERR	Invalid conference handle.

Example

```
HCONF hConf;

if (vocBreakConference(hConf) != E_OK) {
    /* Process error */
}
```

vocClearDT

This function clears the DTMF queue of the specified channel.

Syntax

WORD vocClearDT(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Example

```
HCHN hChn;

if (vcoClearDT(hChn) != E_OK) {
    // Process error
}
```

vocCloseChn

This function closes an opened channel.

Syntax

```
WORD vocCloseChn(
    HCHN hChn
    );
```

Parameters

hChn
Identifies the channel handle.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

If a channel handle opened by the ***vocOpenChn()*** function is no more in use, application should call this function to release the channel for other applications.

Example

```
HCHN  hChn;

if (vocCloseChn(hChn) != E_OK) {
    // Process error
}
```

vocCutWaveFile

This function truncates the file size with input time period.

Syntax

LONG vocCutWaveFile(
LPSTR szVocFile,
WORD wCutTime
);

Parameters

szVocFile

Points to an ASCIIZ string named the voice file to truncate.

wCutTime

Specifies the time period to truncate from the ending of voice file. (Time units is ms)

Return Values

If function was successful, this function returns the remaining time (in seconds) to play of the truncated file. Otherwise, it returns E_ERR.

Example

```
// Cut off the voice file with 5 seconds
if (vcoCutWaveFile("wellcome.wav" , 5000) == E_ERR) {
    // Process error
}
```

vocDelFromConference

This function removes a channel from the specified conference call.

Syntax

```
WORD vocDelFromConference(
    HCONF hConf,
    HCHN hChn
    );
```

Parameters

hConf

Identifies the conference handle.

hChn

Identifies the channel handle to remove.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_CONFERR	Invalid conference handle.

Example

```
HCONF hConf;
HCHN  hChn;

if (vocDelFromConference(hConf, hChn) != E_OK) {
    /* Process error */
}
```

vocDial

This function dials an ASCIIZ string on the specified channel and optionally monitors the call.

Syntax

```
WORD vocDial(
    HCHN hChn,
    LPSTR szPhoneNo,
    WORD wRingCnt,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

szPhoneNo

Points to a null-terminated character string that specifies the dial string.

wRingCnt

Specifies the number of ringback tone to wait after dialed.

wMode

Specifies the running mode and dialing options. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose oen or more:

Value	Meaning
DM_PULSE	Sets this option to use pulse dialing. The default dialing type is touch-tone dialing (DTMF).
DM_BLIND	Sets this option for a blind dialing. The blind dialing option will cause the driver don't check the dial tone before call out the phone number. The default dialing mode will check the dial tone. If no dial tone is defined by the DIAG32.EXE program, any tone on the line will be recognized as a dial tone, To have precise dial tone detection, please define the dial tone frequency to the "Call Progress Monitor Settings" of DIAG32.EXE program.
DM_NOCPA	Sets this option to disable the call progress analysis. The default mode is to enable the call progress analysis.
DM_20PPS	Sets this option for a 20 PPS pulse dialing. The default pulse dialing is 10 PPS. This option must be set with DP_PULSE option.
DM_INTLCPA	Sets this option to force the call progress analysis to use intelligent mode.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	function completed. Call ***vocGetCAR()*** function to get detail information of

	Call Analysis Result.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.
EVT_ERR	I/O Device error. Call ***vocGetLastErr()*** function to get detail error code.

Remarks

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

This function will automatically take the channel off-hook before dialing, if it is in the On-hook State. When the function is complete, the calling channel will remain in the Off-hook State.

All the call analysis parameters and dial parameters are configured by the "Call Progress Monitor Settings" function of DIAG32.EXE program.

The valid characters for pulse dialing and touch-tone dialing are listed below: (characters are case sensitive)

Valid characters for tone dialing:

0 - 9, *, #, A, B, C, D

Valid characters for pulse dialing:

0 - 9

Control characters:

Control Characters	Description
,	Pause. The pause time is set by DIAG32.EXE.
& or !	Make a hook flash. The flash time is set by DIAG32.EXE.
T	Change dial mode to tone dialing.
P	Change dial mode to pulse dialing. (10 PPS)
J	Change dial mode to pulse dialing. (20 PPS)
W	Waiting for a dial tone.
~	Call a pager.

To make a call from the inside PBX, the dialing string should have a prefix string "***9,***" or "***0,***" *before phone number*.

To call a pager, the dialing string should have the following format:

PagerNumber~DisplayNumber

The control character "~" is used to detect the pager tone. To make this function work properly, please define the pager tone at the "Call Progress Monitor Settings" function of DIAG32.EXE program.

The control character "W" is used to detect the dial tone. If DIAG32.EXE program don't define dial tone, any tone on the line will be recognized as a dial tone. To have precise dial tone detection, please define the dial tone frequency to the "Call Progress Monitor Settings" of DIAG32.EXE program.

Example

Example 1: Using function in synchronous model.

```
HCHN  hChn;
CAR Car;

// Call 2195499 with call progress monitor
if (vocDial(hChn, " 2195499", 5, DM_SYNC) != E_OK) {
    // Process error
}
// Call vocGetCAR to get Call Analysis Result.
vocGetCAR(hChn, Car);
switch (Car.wCarType) {
    case CAR_CONNECT:
        :
        break;
    case CAR_NOANSWER:
        :
        break;
    case CAR_BUSY:
        :
        break;
    :
    }
```

Example 2: Using function in asynchronous model.

```
HCHN  hChn;
EVTBLK Event;

// Call 2195499 with call progress monitor
if (vocDial(hChn, " 2195499", 5, DM_ASYNC) != E_OK) {
    // Process error
}
// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);

// Call vocGetCAR to get Call Analysis Result.
vocGetCAR(hChn, Car);
switch (Car.wCarType) {
    case CAR_CONNECT:
        :
        break;
    case CAR_NOANSWER:
        :
        break;
    case CAR_BUSY:
        :
        break;
    :
    }
```

vocEnumChn

This function returns the total channels provided by device driver.

Syntax

WORD vocEnumChn();

Return Values

The return value specifies the total channels provided by device driver.

Example

```
WORD  wChnCnt = vocEnumChn();
printf("There are %d channels in your system", wChnCnt);
```

vocFlashHook

This function flashes the hook state of the specified channel.

Syntax

```
WORD vocFlashHook(
    HCHN hChn,
    WORD wTime,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

wTime

Specifies the flash time, the time unit is ms.

wMode

Specifies the running mode. This parameter can be one of the following values:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	Flash hook function completed.

Remarks

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN hChn;
```

```
// Make a hook flash. The flash time is 200ms.
if (vocFlashHook(hChn, 200, DM_SYNC) != E_OK) {
    // Process error
}
```

Example 2: Using function in asynchronous model.

```
HCHN  hChn;
EVTBLK Event;

// Process error
// Make a hook flash. The flash time is 200ms.
if (vocFlashHook(hChn, 200, DM_ASYNC) != E_OK) {
    // Process error
}
//Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);
```

vocGetCallerID

This function returns the Caller ID message.

Syntax

```
WORD vocGetCallerID (
    HCHN hChn,
    WORD wType,
    LPSTR pMsg
    );
```

Parameters

hChn

Identifies the channel handle.

wType

Specifies the message type to be returned. This parameter can be one of the following values:

Values	Meaning
CID_CALLID	Get the caller's phone number.

pMsg

Pointers to a buffer to receive the requested Caller ID message which is null terminated. The format of returned data has a different definition for each message type:

CID_ALL — Retrieve all caller ID information sent from the CO. The maximum length is 258 bytes; includes header and length byte at the beginning.

CID_GENERAL — The returned null-terminated ASCIIZ string contains the date and time (20 bytes - formatted with "/" and ":" characters; padded with spaces), caller's phone number or reason for absence (20 bytes - padded with spaces), caller name or reason for absence (variable length ≥ 0; no padded)

01234567890123456789 01234567890123456789 01234567890123456789

Date and Time (20 bytes)	Phone Number (20 bytes)	Name (variable length)
03/02*b***11:20***bbbbbbbbb*	**22195499***bbbbbbbbbbbb*	**MICHELLE***b***LEE**Æ
03/02*b***11:20***bbbbbbbbb*	**22195499***bbbbbbbbbbbb*	**P**Æ
03/02*b***11:20***bbbbbbbbb*	**P***bbbbbbbbbbbbbbbbbbb*	**P**Æ
03/02*b***11:20***bbbbbbbbb*	**P***bbbbbbbbbbbbbbbbbbb*	Æ
03/02*b***11:20***bbbbbbbbb*	**O***bbbbbbbbbbbbbbbbbbb*	Æ

b=Blank Æ=Null **P**=Private **O**=Unavailable

CID_CALLID — Get the caller's phone number. A null-terminated ASCIIZ string will be returned. The maximum length of caller's phone number is 300 (MAX_CID_LENGTH) bytes.

pMsg

Pointers to a buffer to receive the requested Caller ID message. The format of returned data has a different definition for each message type.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

This function works only for voice boards that support Caller ID features.

Example

```
HCHN  hChn;
char  buf[MAX_CID_LENGTH];

if (vocGetCallerID(hChn, CID_CALLERID, buf) == E_OK) {
    // Process error
}
//A Caller ID string is returned.
printf("\n\rCaller ID = %s" , buf);
```

vocGetCAR

This function retrieves the call analysis result of the specified channel.

Syntax

WORD vocGetCAR(
HCHN hChn,
LPCAR lpCAR
);

Parameters

hChn

Identifies the channel handle.

lpCAR

Points to a CAR structure to receive the call analysis result. The CAR structure has the following form:

```
typedef struct tagCAR {
  WORD   wCarType;
  DWORD  dwAnsSize;
  WORD   wConnectType;
} CAR;
```

The detail description of this structure is listed below:

wCarType

Specifies the call analysis result. This parameter can be one of the following values:

Value	Meaning
CAR_CONNECT	Indicates the line is connected or the dialing function is completed.
CAR_NOANSWER	No answer.
CAR_BUSY	Line is busy.
CAR_NODIALTONE	No dial tone detected.
CAR_FAX	Called is a fax machine.
CAR_MODEM	Called is a modem unit.
CAR_SWERR	Internal error.
CAR_HWERR	Internal error.

dwAnsSize

Specifies the answer size if line is connected.

w*ConnectType*

Specifies the reason for connect. The connection can be one of the following reasons:

Value	Meaning
CON_CADENCE	Out of tone cadence.
CON_NOISE	Unknown tone detected.
CON_LCDROP	Loop current drop detected.
CON_LCREV	Loop current reversal detected.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Example

```
HCHN  hChn;
CAR  CARInfo;

if (vocDial(hChn, " 2195499", 5, DM_SYNC) != E_OK) {
    // Process error
}
```

```
// Call vocGetCAR to get Call Analysis Result.
if (vocGetCAR(hChn, &CARInfo) != E_OK) {
    // Process error
}
switch (Car.wCarType) {
    case CAR_CONNECT:
        :
        break;
    case CAR_NOANSWER:
        :
        break;
    case CAR_BUSY:
        :
        break;
    :
    }
```

vocGetChnCaps

This function retrieves the channel capability.

Syntax

```
WORD vocGetChnCaps(
    HCHN hChn
    );
```

Parameters

hChn

Identifies the channel handle.

Return Values

Returns the channel capability. This return value can be a combination of the following values:

Value	Meaning
CAP_LOCRECORD	Indicates the channel can record voice from the local phone set.
CAP_SCANFAX	Indicates the channel can support scan fax function. This function only supplies the local phone jack a DC power. User can connect the line of fax machine to the local phone jack and press the start button of fax machine to start transmitting fax. At the same time, a fax function should be invoked to receive fax from this channel.
CAP_VOLUME	Indicates the channel supports volume control.
CAP_BRIDGE	Indicates the channel supports conference function.
CAP_MASTER	Indicates the channel is a master board.
CAP_CALLERID	Indicates the channel supports caller ID detection (only for 4R or 8R card).
CAP_HANDSET	Indicates the channel support handset detection (only for 4R or 8R card).

Example

```
HCHN  hChn;
WORD  wChnCaps;

wChnCaps = vocGetChnCaps(hChn);
if (wChnCaps&CAP_LOCRECORD) {
    // channel supports record function
}
if (wChnCaps&CAP_SCANFAX) {
    // channel supports scan fax function
}
if (wChnCaps&CAP_VOLUME) {
    // channel supports volume control function
}
```

vocGetChnID

This function returns the channel number of the specified channel handle.

Syntax

WORD vocGetChnID(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values
Returns the channel number. The channel number starts from 0.

Remarks
This function does not check the input parameter. Unexpected errors will occur if an invalid channel handle is input.

Example

```
WORD  wChnID;
HCHN  hChn;

vocOpenChn(&hChn, ANY_CHN, NULL);
wChnID=vocGetChnID(hChn);
```

vocGetChnIO

This function retrieves the I/O status of the specified channel.

Syntax

```
WORD vocGetChnIO(
    HCHN hChn
    );
```

Parameters

hChn
Identifies the channel handle.

Return Values

The return value is a combination of the following values:

Value	Meaning
ST_SPKON	Indicates speaker is now on, otherwise the speaker is off.
ST_OFFHOOK	Indicates channel is now off-hook, otherwise is on-hook (Hang up).
ST_AUTORSTON	Indicates the auto-reset function is enabled.
ST_TONE	Indicates the channel is not in silence now.
ST_HANDSET	Indicates the handset is pick up (only for 4R or 8R card).

Example

```
HCHN  hChn;
WORD  wChnIOStatus;

wChnIOStatus = vocGetChnIO(hChn);
if (wChnIOStatus & ST_SPKON) {
    // The speaker is now on.
}
else {  // The speaker is now off.
}
```

vocGetConfGTD

This function retrieves the detailed information of EVT_GTD event for a conference handle.

Syntax

```
WORD vocGetConfGTD(
    HCONF hConf,
    LPGTDDESC lpGTDDesc
    );
```

Parameters

hConf

Identifies the conference handle.

lpGTDDesc

Points to a **GTDDESC** structure to receive information of the detected signal. The **GTDDESC** structure has the following form:

```
typedef struct tagGTDDesc {
  WORD wSignalType;
  WORD wDetectTime;
  WORD wToneID;
} GTDDESC;
```

The detail description of this structure is listed below:

wSignalType

Specifies the signal type. This parameter can be one of the following values:

Value	Meaning
SN_SIL	Indicates a long silence period is detected.
SN_NONSIL	Indicates a long non-silence period is detected.
SN_HANGUP	Indicates a hang-up tone is detected.
SN_LCDROP	Indicates a loop current drop is detected.
SN_LCREV	Indicates a loop current reversal is detected.
SN_SIT1	Indicates a SIT1 tone is detected.
SN_SIT2	Indicates a SIT2 tone is detected.
SN_SIT3	Indicates a SIT3 tone is detected.
SN_USER	A user-defined tone is detected and the ***wToneID*** contains the Tone ID.

wDetectTime

Specifies the time period to detect signal. (Time units is ms)

wToneID

This field contains the Tone ID if the ***wSignalType*** field is SN_USER.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

All the GTD settings are defined by the "Caller Hang-up Settings" of DIAG32.EXE program. If the voice application is running under a different environment, user can redefine the GTD settings via DIAG32 program, and don't need to change the program code.

Example

```
HCONF hConf;
GTDDESC GTD;
if (vocGetConfGTD(hConf, &GTD) != E_OK) {
    // Process error
}
```

vocGetConfVol

This function returns the current volume level of the specified conference call.

Syntax

```
WORD vocGetConfVol(
    HCONF hConf
    );
```

Parameters

hConf
Identifies the conference handle.

Return Values
The return value is from 0 to 15.

Example

```
HCONF hConf;

printf("Current volume level is %u" , vocGetConfVol(hConf)) ;
```

vocGetCurPos

This function retrieves the current playing or recording position.

Syntax

WORD vocGetCurPos(
HCHN hChn,
LPPOSINFO lpPosInfo
);

Parameters

hChn

Identifies the channel handle.

lpPosInfo

Points to a **POSINFO** structure to receive information of the current playing or recording position. The **POSINFO** structure has the following form:

```
typedef struct tagPosInfo {
  DWORD dwPos;
  WORD wTime;
  WORD wSample;
} POSINFO;
```

The detail description of this structure is listed below:

dwPos

Specifies the current position in byte.

w*Time*

Specifies the current position in second.

w*Sample*

Specifies the current sampling rate.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_FUNERR	No voice is playing or recording.

Example

```
HCHN  hChn;
POSINFO  PosInfo;
EVTBLK Event;

// Play back the " voice.wav"  file from the starting point
if (vocPlayFile(hChn, " voice.wav" , 0L, 0, DM_ASYNC) != E_OK) {
    // Process error
}

/* Use vocWaitEvent() to wait for the completion of vocPlayFile() */
while (1) {
    if (vocWaitEvent(hChn, &Event, 100) != E_OK) break;
    vocGetCurPos(hChn, &PosInfo);
    printf(" \rPlaying %lu second(s)" , PosInfo.wTime);
}
```

vocGetDeviceID

This function retrieves the waveform device ID of the specified channel.

Syntax

WORD vocGetDeviceID(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values
Returns the waveform device ID.

Remarks
This waveform device ID returned by this function is used for ***waveOutOpen()*** and ***waveInOpen()*** functions which are described by Windows Multimedia Reference. The voice driver of this voice board is a waveform device, programmers can also use the Low-Level Waveform Audio Services to play back voice and record voice.

Example

```
WORD  wDevID;
HCHN  hChn;
HWAVE  hWave;
WAVEFORMATEX  PcmFmt;

wDeviceID = vocGetDeviceID(hChn);
waveOutOpen(&hWave, wDevID, &PcmFmt, 0, 0, WAVE_MAPPED|CALLBACK_FUNCTION);
// hWave is an WAVEOUT handle now and available for other waveOutXXX function calls.
```

vocGetDT

This function collects digits from the channel' s DTMF queue.

Syntax

```
WORD vocGetDT(
    HCHN hChn,
    LPSTR lpBuf,
    WORD wMaxCnt,
    DWORD dwMaxTime,
    WORD wTermDTs,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

lpBuf

Points to a buffer to receive the digits from DTMF queue. A NULL-terminated ASCIIZ string will be returned. The valid DTMF characters are 1,2,3,4,5,6,7,8,9,0,*,#,A,B,C,D. The length of input buffer must be larger than (number of wMaxCnt) + 1.

wMaxCnt

Specifies the maximum number of digits to receive. If the number of received digits is greater than this parameter, an EVT_MAXDTMF event will be generated.

dwMaxTime

Specifies the time-out interval, in milliseconds. If this parameter is zero, this function will put all the received digits into buffer and return immediately. If this parameter is WT_INFINITE, the function's time-out interval never elapses. If the interval elapses and no enough digit is received, an EVT_MAXTIME event will be generated.

wTermDTs

Specifies the termination digits, in which the reception of any digit will terminate this function. This parameter can be a combination of the following values:

DT_0 : Digit 0.	**DT_6 :** Digit 6.	**DT_A :** Digit A.
DT_1 : Digit 1.	**DT_7 :** Digit 7.	**DT_B :** Digit B.
DT_2 : Digit 2.	**DT_8 :** Digit 8.	**DT_C :** Digit C.
DT_3 : Digit 3.	**DT_9 :** Digit 9.	**DT_D :** Digit D.
DT_4 : Digit 4.	**DT_S :** Digit *.	**DT_ALL :** For all digits.
DT_5 : Digit 5.	**DT_P :** Digit #.	**DT_NONE**: No termination digit.

wMode

Specifies the running mode. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one or more:

Value	Meaning
DM_NOGTD	Set this option to ignore GTD event detection, the default mode will detect GTD event. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD event.
DM_DISTURB	A disturb tone will play automatically while function is waiting for user's DTMF input. To use this option, the specified channel must not in the playing or recording mode.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_MAXDTMF	Maximum number of DTMF digits received.
EVT_IDDTIME	Inter-digit delay time is reached.
EVT_MAXTIME	Maximum function time is reached.
EVT_TERMDT	Terminated by input digit. Call ***vocGetTermDT()*** function to get the terminated digit.
EVT_GTD	GTD Tone detected. Call **vocGetGTD()** function to get detail information.
EVT_STOP	Stopped by ***vocStopChn()*** function.

Remarks

If **DM_DISTURB** option is selected, The volume level of disturb tone can be adjusted by changing the parameter of DSPCMD.INI file, which is in Windows directory. The format is as follow:

[Voclib]
DisturbVol = xx

The ***xx*** value specifies the volume level of disturb tone. The valid value is from 0 to 40, and default is 30. The value 1 is for the maximum volume level and value 40 is for the minimum volume level of disturb tone. Set this value to 0 to disable the disturb function.

If the DTMF detection does not work properly for the long distance phone calls, try to increase the ***DisturbVol*** value to generate the disturb tone with a lower volume.

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN  hChn;
Char  Buf[11];

// Waiting for 10 digits in 5 seconds and define # is the termination digit.
if (vocGetDT(hChn, Buf, 10, 5000, DT_P, DM_SYNC) != E_OK) {
    // Process error
}
```

Example 2: Using function in asynchronous model.

```
HCHN  hChn;
Char  Buf[11];
EVTBLK  Event;

// Waiting for 10 digits in 5 seconds and define # is the termination digit.
if (vocGetDT(hChn, Buf, 10, 5000, DT_P, DM_ASYNC) != E_OK) {
    // Process error
}
//Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);
```

vocGetGTD

This function retrieves the detail information of EVT_GTD event.

Syntax

```
WORD vocGetGTD(
    HCHN hChn,
    LPGTDDESC lpGTDDesc
    );
```

Parameters

hChn

Identifies the channel handle.

lpGTDDesc

Points to a **GTDDESC** structure to receive information of the detected signal. The **GTDDESC** structure has the following form:

```
typedef struct tagGTDDesc {
  WORD  wSignalType;
  WORD  wDetectTime;
  WORD  wToneID;
} GTDDESC;
```

The detail description of this structure is listed below:

wSignalType

Specifies the signal type. This parameter can be one of the following values:

Value	Meaning
SN_SIL	Indicates a long silence period is detected.
SN_NONSIL	Indicates a long non-silence period is detected.
SN_HANGUP	Indicates a hang-up tone is detected.
SN_LCDROP	Indicates a loop current drop is detected.
SN_LCREV	Indicates a loop current reversal is detected.
SN_SIT1	Indicates a SIT1 tone is detected.
SN_SIT2	Indicates a SIT2 tone is detected.
SN_SIT3	Indicates a SIT3 tone is detected.
SN_USER	A user-defined tone is detected and the ***wToneID*** contains the Tone ID.

wDetectTime

Specifies the time period to detect signal. (Time units is ms)

wToneID

This field contains the Tone ID if the ***wSignalType*** field is SN_USER.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

All the GTD settings are defined by the "Caller Hang-up Settings" of DIAG32.EXE program. If the voice application is running under a different environment, user can redefine the GTD settings via DIAG32 program, and don't need to change the program code.

Example

```
HCHN  hChn;
GTDDESC  GTD;
if (vocGetGTD(hChn, &GTD) != E_OK) {
    // Process error
}
```

vocGetLastCST

This function returns the last CST (Channel Status Transition) of the specified channel.

Syntax

WORD vocGetLastCST(
HCHN hChn,
LPCSTBLK pCst
);

Parameters

hChn

Identifies the channel handle.

pCst

Points to a CSTBLK structure to receive the channel status information. The CSTBLK structure has the following form:

```
typedef struct tagCstBlk {
  HCHN hChn;
  WORD wStatus;
  DWORD dwData;
  BYTE bReserved[12];
} CSTBLK;
```

The detail information of this structure is described at ***vocWaitCST()*** function.

Return Values

Returns zero (E_OK) if a CST event was received successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

This function is used to get the last channel status. The status transition event can be masked on or off by calling ***vocSetCSTMask()*** function.

Example

```
HCHN hChn;
CSTBLK CST;

if (vocWaitCST(hChn, &CST, WT_INFINITE) != E_OK) {
    // Process error
```

vocGetLastErr

This function returns the last error code of the specified channel.

Syntax

WORD vocGetLastErr(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values

This function returns the last error code value. See the definition of Function Error Code for detail.

Example

```
HCHN  hChn;

printf("\n\rThe last Error code is %x" , vocGetLastErr(hChn));
```

vocGetLastEvent

This function returns the last termination event block of the specified channel.

Syntax

WORD vocGetLastEvent(
HCHN hChn,
LPEVTBLK pEvent
);

Parameters

hChn

Identifies the channel handle.

pEvent

Points to an EVTBLK structure to receive the event information. The EVTBLK structure has the following form:

```
typedef struct tagEvtBlk {
  HCHN hChn;
  WORD wEvent;
  WORD wTermFun;
  BYTE bReserved[12];
} EVTBLK;
```

The detail description of this structure is described at ***vocWaitEvent()*** function.

Return Values

Returns zero (E_OK) if the last termination event was received successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

See the definition of Termination Event for detail.

Example

```
HCHN  hChn;
EVTBLK Event;

vocGetLastEvent(hChn, &Event);
printf("\n\rThe terminate event code : %u" , Event.wEvent);
```

vocGetLastTerm

This function returns the last termination event of the specified channel.

Syntax

WORD vocGetLastTerm(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values

This function returns a termination event code to indicate the reason for the last termination on the channel. The termination event code is set when an I/O function completed. See the definition of termination event for detail.

Example

```
HCHN  hChn;

printf("\n\rThe terminate event code : %u" , vocGetLastTerm(hChn));
```

vocGetSerialNo

This function returns the serial number of the specified channel.

Syntax

DWORD vocGetSerialNo(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values

Returns the serial number of the specified channel. The return value will be zero if no voice board is found or invalid channel handle is specified.

Example

```
HCHN  hChn;
DWORD  dwChnSerNo;

if ((dwChnSerNo = vocGetSerialNo(hChn)) == 0) {
    printf(" \n\rNo voice board found or invalid channel handle is specified ." );
}
else {
    printf(" \n\rThe serial number of this channel is %lu" , dwChnSerNo);
}
```

vocGetTermDT

This function returns the last terminated digit of the specified channel.

Syntax

```
BYTE vocGetTermDT(
    HCHN hChn
    );
```

Parameters

hChn
Identifies the channel handle.

Return Values

This function returns the last terminated digit. The valid DTMF characters are 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, *, #, A, B, C, D.

Example

```
HCHN  hChn;

printf("\n\rThe terminated digit is %c" , vocGetTermDT(hChn));
```

vocGetVolume

This function retrieves the current output/input volume level of the specified channel.

Syntax

WORD vocGetVolume(
HCHN hChn,
WORD wType
);

Parameters

hChn

Identifies the channel handle.

wType

Specifies the volume type. This parameter can be one of the following values:

Value	Meaning
OUT_VOLUME	To retrieve the current output volume level of voice playing.
IN_VOLUME	To retrieve the current input volume level of voice recording.

Return Values

The return value is from 0 to 15.

Example

```
HCHN  hChn;
WORD  wVolIn, wVolOut;

wVolIn = vocGetVolume(hChn, IN_VOLUME);
wVolOut = vocGetVolume(hChn, OUT_VOLUME);
printf("\n\rChannel volume level In Volume=%u, Out Volume=%u" , wVolIn, wVolOut);
```

vocGetXmitSlot

This function returns SCbus time slot number of voice transmit channel. It returns the SCbus time slot information contained in a TSINFO structure that includes the number of the SCbus time slot connected to the voice transmit channel on a PLUS-4LVSC board.

Syntax

```
WORD vocGetXmitSlot (
    HCHN hChn,
    LPTSINFO lpTSInfo
    );
```

Parameters

hChn

Identifies the channel handle.

lpTSInfo

Points to a TSINFO structure to receive timeslot number in ***pTsArray***. The TSINFO structure has the following form:

```
typedef struct tagTSINFO {
  DWORD  nTsCnt;
  long  *pTsArray;
} TSINFO;
```

The detail description of this structure is listed below:

nTsCnt

This parameter must be initialized with the number of SCbus time slots, typically 1.

pTsArray

This parameter must be initialized with a pointer to a valid array. Upon return from the function, the array will contain the number (between 0 and 1023) of the SCbus time slot on which the voice channel transmits.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_BADPARAM	Invalid input parameter. The ***nTsCnt*** parameter is not equal to 1.
E_BADTYPE	Invalid channel type (voice, analog, fax ..etc)
E_FUNERR	Function is not supported in current bus configuration.

Remarks

A voice channel on a PLUS-4LVSC board can transmit on only one SCbus time slot.

Example

```
HCHN  hChn;
TSINFO TsInfo;

TsInfo.nTsCnt = 1;   //the number of timeslot to get is one
if (vocGetXmitSlot(hChn, &TsInfo) == E_OK) {
    printf(" \n\rThe transmit time slot for voice=%d" , TsInfo.pTsArray[0]);
}
```

vocInitDriver

This function initializes the voice driver.

Syntax

WORD vocInitDriver();

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_DRVERR	Fail to initialize driver.
E_NOCHN	No channel found.
E_NOMEM	Not enough memory.

Remarks

Application should call this function before call other APIs.

Example

```
if (vocInitDriver() != E_OK) {
    // Process error
}
```

vocIsLineConnect

This function is used to detect whether a CO line or PBX line is connected to the specified channel.

Syntax

BOOL vocIsLineConnect(
HCHN hChn
);

Parameters

hChn

Identifies the channel handle.

Return Values

Returns TRUE if the CO or PBX line is connected. Otherwise, it returns FALSE.

Remarks

To on-hook the specified channel before call this function. This function will take about 2 seconds to detect the line status, and the channel will be in the on-hook state after function completed.

Example

```
if (vocIsLineConnect()) {
    printf(" \n\rThe CO line is connected" );
}
else {
    printf(" \n\rThe CO line is not connected" );
}
```

vocListen

This function connects voice listen channel to SCbus time slot. This function uses the information stored in the TSINFO structure to connect the receive voice (listen) channel on a PLUS-4LVSC board to an SCbus time slot.

Syntax

```
WORD vocListen (
   HCHN hChn,
   LPTSINFO lpTSInfo
   );
```

Parameters

hChn

Identifies the channel handle.

lpTSInfo

Points to a TSINFO structure to specify the time slot information.. The TSINFO structure has the following form:

```
typedef struct tagTSINFO {
  DWORD  nTsCnt;
  long  *pTsArray;
} TSINFO;
```

The detail description of this structure is listed below:

nTsCnt

This parameter must be set to 1.

pTsArray

This parameter must be initialized with a pointer to a valid array. The first element of this array must contain a valid SCbus time slot number (between 0 and 1023) which was obtained by issuing an ***anaGetXmitSlot()*** function. Upon return from this function, the voice receive channel will be connected to the SCbus time slot.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_BADPARAM	Invalid input parameter. The ***nTsCnt*** parameter is not equal to 1.
E_BADTYPE	Invalid channel type (voice, analog, fax ..etc)
E_FUNERR	Function is not supported in current bus configuration.

Remarks

This function sets up a half-duplex connection. For a full-duplex connection, the receive (listen) channel of the other device must be connected to the voice transmit channel.

Although multiple voice channels may listen (be connected) to the same SCbus time slot, the receive of a voice channel can connect to only one SCbus time slot.

Calling the ***vocListen()*** function to connect to a different SCbus time slot will automatically break an existing connection. Thus, when changing connections, you need not call the ***vocUnlisten()*** function.

Example

```
HCHN hChn;
TSINFO TsInfo;
TsInfo.nTsCnt = 1;
TsInfo.pTsArray[0] = 512; // if the timeslot going to listen is 512
```

vocListen( )

```
if (vocListen(hChn, &TsInfo) == E_OK) {
    printf(" \n\rVoice channel is ready to listen time slot 512!");
}
```

vocMakeConference

This function makes a conference call for all the input channels.

Syntax

```
WORD vocMakeConference(
    LPHCHN lphChnList,
    WORD wVolume,
    LPHCONF lphConf,
    WORD wMode
    );
```

Parameters

lphChnList

Points to a buffer to specify all the channel handles, which are included in a conference call. This input array is terminated with zero.

wVolume

Specifies the default volume level for this conference call. The valid value is from 0 to 15. The recommended volume level is 8, and 15 for the biggest level.

lphConf

Points to a buffer to receive the conference handle.

wMode

Specifies the options. This parameter can be a combination of the following va lues:

Value	Meaning
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryConfGTD()*** function to retrieve the GTD events.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_CONFFULL	All conference handles are in use.
E_CHNINUSE	One of the input channels is in use for other conference call.
E_TOOMANYCHN	Too many channels in a conference handle.
E_FUNERR	No conference call function supported.
E_SYSERR	Failed to synchronize.
E_NOMEM	Insufficient memory.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	End of conference call, stopped by vocBreakConference().
EVT_GTD	GTD Tone detected. Call **vocGetConfGTD()** function to get detailed information.

Remarks

This function will be running in **asynchronous** mode only.

It will return zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event to indicate the function has completed. The possible termination events are listed above. A returned nonzero value by this function indicates an error occurred.

The maximum number of channels in a conference call is 4 channels. If a conference handle has

occupied a channel, then the channel could not become a member of another conference call before it is removed from the original conference handle

The ***wVolume*** parameter of ***vocMakeConference()*** function defines the default volume level of the specified conference call. This volume level can be adjusted by the ***vocGetConfVol()*** and ***vocSetConfVol()*** functions.

Once a conference call was made, application is able to call ***vocAddToConference()*** or ***vocDelFromConference()*** function to add or remove a channel. All conference calls made by ***vocMakeConference()*** function could be terminated by the ***vocBreakConference()*** function or a GTD event is detected.

The channels in conference call could not be used to play file, record file or dialing out until it is removed from the conference call.

Example

There are two ways to make a conference call:

Example 1: Call this function with a channel handle list.

```
HCHN ChnList[3];
HCHN hChn1, hChn2;
HCONF hConf;

if (vocOpenChn(&hChn1, ANY_CHN, NULL) != E_OK) {../*Error Process*/}
if (vocOpenChn(&hChn2, ANY_CHN, NULL) != E_OK) {../*Error Process*/}

// To make a conference call for channel 1 and 2
ChnList[0] = hChn1, ChnList[1] = hChn2, ChnList[2] = 0;
if (vocMakeConference(ChnList, 8, &hConf, 0) != E_OK) {../*Error Process*/};
   :
   :
while (1) {
if (vocWaitConfEvent(hConf, &wEvent, 500) == E_OK) {//wait event for 500ms
  switch (wEvent) {
  case EVT_END: …                    //terminated by vocBreakConference()
  case EVT_GTD:vocGetConfGTD(hConf, &GTDEvt);   //get detailed GTD event
  default:
  break;
}
if (vocReadDT(hChn1)==' 0' ) vocBreakConference(hConf); //stop conference if channel1 detect DTMF 0 code
}
```

Example 2: Call this function with a null handle list.

Call the *vocMakeConference()* function with a null channel handle list and then call the *vocAddToConference()* or *vocDelFromConference()* function to add or remove from conference members.

```
HCHN ChnList[4];
HCHN hChn1, hChn2, hChn3;
HCONF hConf;

if (vocOpenChn(&hChn1, ANY_CHN, NULL) != E_OK) {../*Error Process*/}
if (vocOpenChn(&hChn2, ANY_CHN, NULL) != E_OK) {../*Error Process*/}
if (vocOpenChn(&hChn3, ANY_CHN, NULL) != E_OK) {../*Error Process*/}

// To make a null conference call
if (vocMakeConference(NULL, 8, &hConf, 0) != E_OK) {../*Error Process*/};
// Add channel 1 to the conference call
if (vocAddToConference(hConf, hChn1) != E_OK) { ../*Error Process*/};
// Add channel 2 to the conference call
```

```
if (vocAddToConference(hConf, hChn2) != E_OK) {../*Error Process*/};
// Add channel 3 to the conference call
if (vocAddToConference(hConf, hChn3) != E_OK) { ../*Error Process*/};
// Remove channel 1 from the conference call
if (vocDelFromConference(hConf, hChn1) != E_OK) {../*Error Process*/};
   :
   :
if (vocWaitConfEvent(hConf, &wEvent, INFINITE) == E_OK) {
  switch (wEvent) {
  case EVT_END:  .//terminated by vocBreakConference()
  case EVT_GTD: vocGetConfGTD(hChn1, &GTDEvt);  //Get detailed GTD event
  }
}
```

vocMonitorChn

This function is used to monitor a channel.

Syntax

WORD vocMonitorChn(
HCHN hChn,
HCHN hChnBeMonitored,
WORD wMode
);

Parameters

hChn

Identifies the channel handle to monitor.

hChnBeMonitored

Identifies the channel handle to be monitored.

wMode

Specifies the running mode and recording mode. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one only:

Value	Meaning
VM_ADPCM	Uses OKI ADPCM format for voice recording. This value cannot be used with VM_MULAW and VM_WAVE options.
VM_MULAW	Uses the μ-law PCM format for voice recording. This value cannot be used with VM_ADPCM and VM_WAVE options.
VM_WAVE	Uses the multimedia wave format for voice recording. This value cannot be used with VM_ADPCM and VM_MULAW options.

Choose one or more:

Value	Meaning
VM_SR6	Uses 6KHz sampling rate for voice recording. If this value is not specified, the default sampling rate is 8KHz.
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD events.

Return Values

Returns zero (E_OK) if a termination event was received successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_BUSY	The specified monitor channel is busy, or the specified channel has been already monitored by another channel.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_ERR	I/O Device error. Call ***vocGetLastErr()*** function to get detail error code.
EVT_GTD	GTD Tone detected. Call **vocGetGTD()** function to get detail information.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.

Remarks

This function is used to monitor the channel (***hChnBeMonitored***), and meanwhile play back the recording data at the specified channel (***hChn***). If program turns on the speaker of ***hChn*** channel, a real-time (0.5 second delay) conversation of ***hChnBeMonitored*** channel can be heard from the speaker of ***hChn*** channel.

When the monitor function is working, the ***hChn*** channel cannot play file, record file or dial out until the monitor function is completed. The ***hChnBeMonitored*** channel also cannot play file or dial out, but it can call ***vocRecordFile()*** function to record voice file during the monitor time.

To stop the monitor function, calls the ***vocStopChn()*** function. It will not stop the recording function of the ***hChnBeMonitored*** channel if it is in the progress of recording file

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN  hChn, hChnBeMonitored;
if (vocMonitorChn ((hChn, hChnBeMonitored , DM_SYNC) != E_OK) {
    // Process error
}
```

Example 2: Using function in asynchronous model.

```
HCHN  hChn, hChnBeMonitored;
EVTBLK Event;
if (vocMonitorChn((hChn, hChnBeMonitored , DM_ASYNC) != E_OK) {
    // Process error
}
// Waiting for termination event.
vocWaitEvent(hChnMon, &Event, WT_INFINITE);
```

vocOpenChn

This function opens a free channel handle for operation.

Syntax

```
WORD vocOpenChn(
  LPHCHN phChn,
  WORD wChnID,
  LPCBDESC pCallBack
  );
```

Parameters

phChn

Points to a channel handle. This location will be filled with a channel handle identifying the open channel device. Use this handle to identify the channel device when calling other functions.

wChnID

Identifies the channel number to open. The valid channel number starts from 0. If the **ANY_CHN** is specified, driver will choose the first free channel.

pCallBack

Points to a CBDESC structure to specify the callback information. If no callback function is required, this value can be zero. The CBDESC structure has the following form:

```
typedef struct tagCBDesc {
  DWORD dwEventCallback;
  DWORD dwEventCallbackInst;
  DWORD dwEventFlag;
  DWORD dwEventMsg;
  DWORD dwCSTCallback;
  DWORD dwCSTCallbackInst;
  DWORD dwCSTFlag;
  DWORD dwCSTMsg;
} CBDESC;
```

The detail description of this structure is listed below:

dwEventCallback

This field contains a window handle or address of a fixed callback function to be called during a termination event occurs. If no callback function is required, this value can be zero. For more information on the callback function, please see Programming Models section.

dwEventCallbackInst

This field contains a user-instance data passed to the function callback mechanism for termination event. This parameter is not used with the window callback mechanism.

dwEventFlag

Specifies the event flag for opening the channel. This field can be one of the following values:

Values	Meaning
CB_FUNCTION	The *dwEventCallback* parameter is a callback procedure address.
CB_WINDOW	The *dwEventCallback* parameter is a window handle.
CB_POLLING	No callback mechanism. This is the default setting. Applcations can call ***vocWaitEvent()*** function to wait for a termination event.

dwEventMsg

Defines the message value for termination event.

dwCSTCallback

This field contains a window handle or address of a fixed callback function to be called during a channel status event occurs.. If no callback function is required, this value can be zero. For more information on the callback function, please see Programming Models section.

dwCSTCallbackInst

This field contains a user-instance data passed to the function callback mechanism for CST event. This parameter is not used with the window callback mechanism.

dwCSTFlag

Specifies the CST flag for opening the channel. This field can be one of the following values:

Values	Meaning
CB_FUNCTION	The *dwCSTCallback* parameter is a callback procedure address.
CB_WINDOW	The *dwCSTCallback* parameter is a window handle.
CB_POLLING	No callback mechanism. This is the default setting. Applications can call ***vocWaitCST()*** function to wait for a CST event.

dwCSTMsg

Defines the message value for CST event.

Return Values

Returns zero (E_OK) if the function was successful and the channel handle is filled in phChn. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_NOCHN	No channel available to be used .
E_ALLOCATED	The specified channel is already allocated.

Remarks

The function is used to allocate a channel and set the programming model for it. There are two programming models in application development: Synchronous and Asynchronous. The programming model of each channel can be changed by calling ***vocSetEventCallback()*** function.

Example

To open a channel without callback programming model:

```
HCHN  hChn;
if (vocOpenChn(&hChn, ANY_CHN, NULL) != E_OK) {
    // Process error
}
```

To open a channel with callback programming model:

```
HCHN  hChn;
CBDESC  CB;

// Set the CB
CB.dwEventCallback = (DWORD)vocCallBackProc;
CB.dwEventCallbackInst = NULL ;
CB.dwEventFlag = CB_FUNCTION;
CB.dwEventMsg = NULL;
CB.dwCSTCallback = NULL;
CB.dwCSTCallbackInst = NULL;
CB.dwCSTFlag = NULL;
CB.dwCSTMsg = NULL;

// Open a channel with callback model.
if (vocOpenChn(&hChn, ANY_CHN, &CB) != E_OK) {
    // Process error
}
```

vocPlayFile

This function plays back one or more recorded voice file on the specified channel.

Syntax

```
WORD vocPlayFile(
    HCHN hChn,
    LPSTR szVocFile,
    DWORD dwStartPos,
    WORD wTermDTs,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

szVocFile

Points to a null-terminated character string named a voice file or voice filelist to play. The voice filelist is a character string containing lots of voice file separated by semi-comma. For example, the input string "VocFile1;VocFile2;VocFile3" will guide this function to play the voice files VocFile1, VocFile2 and VocFile3 sequentially. All voice files of the filelist must have the same media format and the maximum size of filelist is 5120 bytes.

DwStartPos

Specifies the starting position to play back.

WTermDTs

Specifies the termination digits, in which the reception of any digit will terminate this function. This parameter can be a combination of the following values:

DT_0 : Digit 0.	**DT_6 :** Digit 6.	**DT_A :** Digit A.
DT_1 : Digit 1.	**DT_7 :** Digit 7.	**DT_B :** Digit B.
DT_2 : Digit 2.	**DT_8 :** Digit 8.	**DT_C :** Digit C.
DT_3 : Digit 3.	**DT_9 :** Digit 9.	**DT_D :** Digit D.
DT_4 : Digit 4.	**DT_S :** Digit *.	**DT_ALL :** For all digits.
DT_5 : Digit 5.	**DT_P :** Digit #.	**DT_NONE**: No termination digit.

wMode

Specifies the running mode and options. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one or more:

Value	Meaning
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD events.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_NOFILE	Voice file not found.
E_READERR	Failed to read voice file.

E_FMTERR	Unknown format.
E_NOMEM	Insufficient memory.
E_CHNERR	Invalid channel handle.
E_SYSERR	Failed to synchronize

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	End of playing.
EVT_TERMDT	Terminated by input digit. Call ***vocGetTermDT()*** function to get the terminated digit.
EVT_ERR	I/O Device error. Call ***vocGetLastErr()*** function to get detail error code.
EVT_GTD	GTD Tone detected. Call ***vocGetGTD()*** function to get detail information.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.

Remarks

This function will return **E_FMTERR** if the format of specified wave file is not supported. The supported waveforms are:

- 6 and 8KHz, 8-bit Windows PCM
- 6 and 8KHz, 8-bit µ-law PCM
- 6 and 8KHz, 4-bit OKI ADPCM

If the termination event is **EVT_TERMDT**, all the input digits including terminated digit are still in the channel's DTMF queue. Applications can call the ***vocGetDT()*** or ***vocReadDT()*** function to retrieve the input digits or call the ***vocClearDT()*** function to clear the DTMF queue.

Two sampling rates (i.e. 6K or 8K) are provided for playing back voice files, and all channels must apply the same sampling rate. That means application should play back voice files either in 6K or 8K sampling rate for all channels. For example, if the channel 1 is playing in 8K sampling rate, the other channels should also play file in 8K sampling rate.

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN  hChn;
// Play back the "voice.wav"  file from the starting point, and terminated by any input digit
if (vocPlayFile(hChn, "voice.wav", 0L, DT_ALL, DM_SYNC) != E_OK) {
    // Process error
}
```

Example 2: Using function in asynchronous model.

```
HCHN hChn;
```

vocPlayFile( )

```
EVTBLK Event;
// Play back the " voice.wav"  file from the starting point, and terminated by any input digit
if (vocPlayFile(hChn, " voice.wav" , 0L, DT_ALL, DM_ASYNC) != E_OK) {
    // Process error
}
…
/* Use vocWaitEvent() to wait for the completion of vocPlayFile() */
vocWaitEvent(hChn, &Event, WT_INFINITE);
```

vocPlayTone

This function generates a defined tone on the specified channel.

Syntax

WORD vocPlayTone(
HCHN hChn,
LPFREQDESC lpFreqDesc,
WORD wMode
);

Parameters

hChn

Identifies the channel handle.

lpFreqDesc

Points to a FREQDESC structure to input the tone information. The FREQDESC structure has the following form:

```
typedef struct tagFreqDesc {
  int Freq1;
  int Freq2;
  int Amp1;
  int Amp2;
  WORD wDuration;
} FREQDESC;
```

The detail description of this structure is listed below:

Freq1

Specifies the frequency for tone 1. The valid value is from 200 - 4000 and unit is Hz.

Freq2

Specifies the frequency for tone 2. If the playing tone is a dual-frequency, specifies the frequency of tone 2 for this field. If the playing tone is a single frequency, fill this field with zero. The valid value is from 200 - 4000 and unit is Hz.

Amp1

Specifies the amplitude for tone 1. The valid value is from -40 to 0 and unit is dB. The recommended value is DEF_AMP (-10dB).

Amp2

Specifies the amplitude for tone 2. The valid value is from -40 to 0 and unit is dB. The recommended value is DEF_AMP (-10dB).

wDuration

Specifies the duration of tone. The time unit is 10 ms. (-1 means infinite duration)

wMode

Specifies the running mode and options. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one or more:

Value	Meaning
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD events.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	End of tone generation.
EVT_ERR	I/O Device error. Calls ***vocGetLastErr()*** function to get detail error code.
EVT_GTD	A GTD Tone detected. Calls **vocGetGTD()** function to get detail information.

Remarks

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN  hChn;
FREQDESC  FreqD;

// Play a 440,480Hz dual-frequency tone for 1 second.
FreqD.Freq1 = 440, FreqD.Freq2 = 480;
FreqD.Amp1 = FreqD.Amp2 = DEF_AMP;
FreqD.wDuration = 100;
if (vocPlayTone(hChn, &FreqD, DM_SYNC) != E_OK) {
    /* process error */
    }
```

Example 2: Using function in asynchronous model.

```
HCHN  hChn;
FREQDESC  FreqD;
EVTBLK Event;

// Play a 480,620Hz dual-frequency tone for 500 ms.
FreqD.Freq1 = 480, FreqD.Freq2 = 620;
FreqD.Amp1 = FreqD.Amp2 = DEF_AMP;
FreqD.wDuration = 50;
if (vocPlayTone(hChn, &FreqD, DM_ASYNC) != E_OK) {
    /* process error */
    }
// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);
    }
```

vocPutSignal

This function is used to simulate the generation of channel' s signal.

Syntax

WORD vocPutSignal(
WORD wChnID,
LPSIGNALBLK pSignalBlk
);

Parameters

wChnID

Identifies the zero-based channel number.

pSignalBlk

Points to a SIGNALBLK structure to simulate the generation of channel' s signal. The SIGNALBLK structure has the following form:

```
typedef struct tagSignalBlk {
  WORD wSigType;
  DWORD dwData;
  BYTE bReserved[12];
} SIGNALBLK;
```

The detail description of this structure is listed below:

wSigType

Specifies the signal type that is going to simulate on the specified channel. This field can be one of the following values:

Values	Meaning
SIM_RING	Simulate to generate a ring signal.
SIM_DIGIT	Simulate to generate a DTMF tone signal.
SIM_DIAL	Simulate to complete dial function.
SIM_GTD	Simulate to generate Global Tone Detection.

dwData

Contains the data associated with ***wSigType***. This field has a different definition for each event type:

wSigType	dwData
SIM_RING	Specifies the timer tick value when ring signal occurred.
SIM_DIGIT	Specifies the ASCII digit. (0 – 9, *, #, A – D)
SIM_DIAL	Specifies the dialed results, the definition are the same as ***wCarType*** of ***vocGetCAR()*** fucntion.
SIM_GTD	Reserved.(don' t care)

Return Values

Returns zero (E_OK) if a signal simulation is generated successfully. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel number.
E_BADPARAM	Invalid input parameter.

Remarks

This function is used to simulate the generation of channel' s signal just as a real signal occurred. Such as a ring signal detected or DTMF tone is detected. In particular, this function is suitable for those environments that the specified channel is not connected with telephone line and is still capable of demonstrating other voice functions like voice play or recording voice.

Example 1 (Simulate to generate a ring signal)

```
WORD wChnID;
```

```
SIGNALBLK  SigBlk;
{
    /* Simulate one ring occurred */
    SigBlk.wSigType = SIM_RING;
    if (vocPutSignal(wChnID, &SigBlk) != E_OK) { // Process error }
}
```

Example 2 (Simulate to generate a DTMF signal)

```
WORD  wChnID;
SIGNALBLK  SigBlk;
{
    /* Simulate one DTMF tone occurred */
    SigBlk.wSigType = SIM_DIGIT;
    SigBlk.dwData = '0' ;          //Simulate a DTMF 0 tone occurred
    if (vocPutSignal(wChnID, &SigBlk) != E_OK) { // Process error }
}
```

vocQueryConfGTD

Retrieves the GTD event for a conference call.

Syntax

BOOL vocQueryConfGTD(
HCONF hConf
);

Parameters

hConf
Identifies the conference handle.

Return Values

Returns a TRUE(i.e. value 1) if a GTD event is detected. The more detail information about GTD event can be retrieved by calling ***vocGetConfGTD()*** function. Otherwise, it returns a FALSE (i.e. value 0) indicating no GTD event detected. In order to prevent from an unexpected error, the input parameter must be a valid conference handle

Remarks

This function is used to retrieve the GTD event for a conference call. If application has called the ***vocMakeConference()*** function with the **DM_NOGTD** option, it can use this function to retrieve the GTD event.

A GTD event will always terminate a executing function. To prevent from this condition, application can use ***vocSetGTDMask()*** function to limit the allowable GTD events or call functions with **DM_NOGTD** option to ignore the GTD detection.

When the ***vocMakeConference()*** function is called with **DM_NOGTD** option, applcation can continuously use the ***vocQueryConfGTD()*** function to check the GTD events.

Example

```
HCONF  hConf;
GTDDESC  GTD;
EVTBLK  Event;
if (vocMakeConference(NULL, 8, &hConf, DM_NOGTD) != E_OK) {
    //Process making conference error
}
while (1) {
    if (vocQueryConfGTD(hConf) == TRUE) {
        vocGetConfGTD(hConf);
        // Process GTD occurred
    }
    if (vocWaitConfEvent(hConf, &Event, 500) == E_OK) {
        //Process function completion
        break;
    }
}
```

vocQueryGTD

Retrieves the GTD event for some limited functions.

Syntax

BOOL vocQueryGTD(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values

Returns a TRUE(i.e. value 1) if a GTD event is detected. The more detail information about GTD event can be retrieved by calling ***vocGetGTD()*** function. Otherwise, it returns a FALSE (i.e. value 0) indicating no GTD event detected. In order to prevent from an unexpected error, the input parameter must be a valid channel handle

Remarks

This function is used to retrieve the GTD events for the specified channel. If application has called the some asynchronous functions with the **DM_NOGTD** option, it can use this function to retrieve the GTD event.

A GTD event will always terminate a executing function. To prevent from this condition, application can use ***vocSetGTDMask()*** function to limit the allowable GTD events or call functions with **DM_NOGTD** option to ignore the GTD detection.

When an asynchronous function is called with **DM_NOGTD** option, applcation can continuously use the ***vocQueryGTD()*** function to check the GTD events.

Example

```
HCHN  hChn;
GTDDESC  GTD;
EVTBLK  Event;
if (vocPlayFile(hChn, " Greet.wav" , 0, DT_NONE, DM_ASYNC+DM_NOGTD) != E_OK) {
    //Process play file error
}
while (1) {
    if (vocQueryGTD(hChn) == TRUE) {
        vocGetGTD(hChn);
        // Process GTD occurred
    }
    if (vocWaitEvent(hChn, &Event, 500) == E_OK) {
        //Process function completion
        break;
    }
}
```

vocReadDT

This function reads a digit from the channel' s DTMF queue.

Syntax

char vocReadDT(
HCHN hChn
);

Parameters

hChn
Identifies the channel handle.

Return Values

Returns zero if there is no digit in the channel' s DTMF queue. Otherwise, it returns an ASCII character of DTMF digit. The possible returned ASCII characters are 0,1, 2, 3, 4, 5, 6, 7, 8, 9, *, #, A, B, C, D.

Remarks

This function just reads single character from DTMF queue, and there is no termination event will be generated when this function is complete.

Example

```
HCHN  hChn;
EVTBLK Event;

// To play file of voice.wav with no DTMF termination and running asynchronously.
if (vocPlayFile(hChn, "voice.wav", 0L, DT_NONE, DM_ASYNC) != E_OK) {
    // Process error
}
while (TRUE) {
//Waiting for termination event.
    if (vocWaitEvent(hChn, &Event, 1000) == E_OK) {
      …
      break;
    }
    if (vocReadDT(hChn) == ' 0' ) {..//Process for DTMF 0 is detected}
}
```

vocReadOEMVersion

This function returns the OEM version number of the specified channel.

Syntax

```
DWORD vocReadOEMVersion(
    HCHN hChn
    );
```

Parameters

hChn

Identifies the channel handle.

Return Values

Returns the OEM version number of the specified channel. The return value will be zero if no voice board is found or invalid channel handle is specified.

Example

```
HCHN  hChn;
DWORD  dwOEMVersionNo;

if ((dwOEMVersionNo = vocReadOEMVersion(hChn)) == 0) {
    printf(" \n\rNo voice board found or invalid channel handle is specified ." );
}
else {
    printf(" \n\rThe OEM version number of this channel is %lu" , dwOEMVersionNo);
}
```

vocRecordFile

This function records a voice file.

Syntax

```
WORD vocRecordFile(
    HCHN hChn,
    LPSTR szVocFile,
    DWORD dwRecTime,
    WORD wTermDTs,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

szVocFile

Points to a null-terminated character string named the file to record.

dwRecTime

Specifies the time interval for voice recording, in milliseconds. If this parameter is WT_INFINITE, the record time interval never elapses.

wTermDTs

Specifies the termination digits, in which the reception of any digit will terminate this function. This parameter can be a combination of the following values:

DT_0 : Digit 0.	**DT_6 :** Digit 6.	**DT_A :** Digit A.
DT_1 : Digit 1.	**DT_7 :** Digit 7.	**DT_B :** Digit B.
DT_2 : Digit 2.	**DT_8 :** Digit 8.	**DT_C :** Digit C.
DT_3 : Digit 3.	**DT_9 :** Digit 9.	**DT_D :** Digit D.
DT_4 : Digit 4.	**DT_S :** Digit *.	**DT_ALL :** For all digits.
DT_5 : Digit 5.	**DT_P :** Digit #.	**DT_NONE**: No termination digit.

wMode

Specifies the running mode and recording format. This parameter can be a combination of the following values:

Choose one only:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Choose one only:

Value	Meaning
VM_ADPCM	Uses OKI ADPCM format for voice recording. This value cannot be used with VM_MULAW and VM_WAVE options.
VM_MULAW	Uses the μ-law PCM format for voice recording. This value cannot be used with VM_ADPCM and VM_WAVE options.
VM_WAVE	Uses the multimedia wave format for voice recording. This value cannot be used with VM_ADPCM and VM_MULAW options.

Choose one or more:

Value	Meaning
VM_SR6	Uses 6KHz sampling rate for voice recording. If this value is not specified, the default sampling rate is 8KHz.
VM_NOAGC	Records without AGC (Automatic Gain Control). If this value is not specified, the default setting is recording with AGC.

VM_SCOMP	Uses silence compression for voice recording.
DM_NOGTD	Set this option to ignore GTD events. The default mode will detect GTD events. If this option is specified, applications can call ***vocQueryGTD()*** function to retrieve the GTD events.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CREATEERR	Failed to create the voice file.
E_FMTERR	Unknown format.
E_NOMEM	Insufficient memory.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_MAXTIME	The recording time is reached.
EVT_TERMDT	Terminated by input digit. Call ***vocGetTermDT()*** function to get the terminated digit.
EVT_ERR	I/O Device error. Call ***vocGetLastErr()*** function to get detail error code.
EVT_GTD	GTD Tone detected. Call **vocGetGTD()** function to get detail information.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.

Remarks

If the termination event is **EVT_TERMDT**, all the input digits including terminated digit are still in the channel's DTMF queue. Applications can call the ***vocGetDT()*** or ***vocReadDT()*** function to retrieve the input digits or call the ***vocClearDT()*** function to clear the DTMF queue.

Two sampling rates (i.e. 6K or 8K) are provided for the voice recording, and all channels must apply the same sampling rate. That means application should record voice files either in 6K or 8K sampling rate for all channels. For example, if the channel 1 is recording in 8K sampling rate, the other channels should also record file in 8K sampling rate.

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN hChn;

// Record a voice file. Function will be stopped by either 30 seconds reaches, or any input DTMF.
```

```
if (vocRecordFile(hChn, "voice.wav", 30000, DT_ALL, DM_SYNC) != E_OK) {
    // Process error
}
```

Example 2: Using function in asynchronous mode l.

```
HCHN hChn;
EVTBLK Event;
// Record a voice file. Function will be stopped by either 60 seconds reaches, or input digit 0.
if (vocRecordFile(hChn, "voice.wav", 60000, DT_0, DM_ASYNC) != E_OK) {
    // Process error
}
// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);
```

vocSetChnIO

This function controls the I/O status of the specified channel.

Syntax

WORD vocSetChnIO(
HCHN hChn,
WORD wSwitch
);

Parameters

hChn

Identifies the channel handle.

wSwitch

Specifies the I/O control type , this parameter can be one of the following values:

Value	Meaning
IO_SPK_ON	Turns on the speaker of the specified channel.
IO_SPK_OFF	Turns off the speaker of the specified channel.
IO_PHONE_CO	Connects the local phone set to the CO line.
IO_PHONE_PC	Connects the local phone set to the PC system.
IO_RINGCTRL_ON	Enables the ring detection for fax/modem chip.
IO_RINGCTRL_OFF	Disables the ring detection for fax/modem chip.
IO_AUTORST_ON	Enables the auto-reset function of the specified channel.
IO_AUTORST_OFF	Disables the auto-reset function of the specified channel.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	**Meaning**
E_CHNERR	Invalid channel handle.

Remarks

To record voice from the local phone set, call this function with **IO_PHONE_PC** parameter. After voice recorded, the local phone set should be in the **IO_PHONE_CO** status.

For each voice board, if one of the channel speakers is turned on, the other channels' speaker, which are on the same voice board will be turned off automatically.

The **IO_PHONE_CO and IO_PHONE_PC** are only available for the first port of voice board.

Example

```
HCHN  hChn;

// Turn on the speak of the specified channel
if (vocSetChnIO(hChn, IO_SPK_ON) != E_OK) {
    // Process error
}
```

vocSetChnParam

This function sets the channel parameters.

Syntax

```
WORD vocSetChnParam(
    HCHN hChn,
    WORD wParam,
    DWORD dwData
    );
```

Parameters

hChn

Identifies the channel handle.

wParam

Specifies the channel parameter to change. This parameter can be one of the following values:

Values	Meaning
CP_IDDTIME	Set the Inter-digit delay (IDD) time.
CP_IDDFLAG	Set the IDD options.
CP_CALLERID	Enable or disable the Caller ID function (only for 4R or 8R card).
CP_HANDSET	Enable or disable the handset detection (only for 4R or 8R card).

dwData

Contains the data associated with ***wParam***. This field has a different definition for each channel parameter:

wParam	dwData
CP_IDDTIME	Specifies the IDD time in seconds. A nonzero value will enable the IDD time check function, and a zero value will disable the IDD time check function. The default IDD time is 0 after channel opened.
CP_IDDFLAG	1 – indicates the IDD time check is starting from the first digit. 0 – indicates the IDD time check is starting from the second digit. This is the default setting after channel opened.
CP_CALLERID	This parameter is only available on Plus-4R and Plus-8R voice boards, and the **dwData** field can be one of the following value: CPX_FSK – To enable the Caller ID detection by using FSK demodulation (compatible with the Bell-202 specification, 1200 baud rate FSK signal) CPX_DTMF – To enable the Caller ID detection by using DTMF method. CPX_DISABLE – To disable the detection of Caller ID function. This is the default setting after channel opened.
CP_HANDSET	This parameter is only available on Plus-4R and Plus-8R voice boards, and the **dwData** field can be one of the following value: CPX_ENABLE – To enable the detection of handset function. CPX_DISABLE – To disable the detection of handset function.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_ERR	Requested function is not supported.

Example:

```
HCHN  hChn;

// Set the IDD time to 2 seconds
if (vocSetChnParam(hChn, CP_IDDTIME, 2) != E_OK) {
    // Process error,
}
```

vocSetConfVol

This function controls the volume level of the specified conference call.

Syntax

```
WORD vocSetConfVol(
    HCONF hConf,
    WORD wVolume
    );
```

Parameters

hConf

Identifies the conference handle.

wVolume

Specifies the volume level to set. The valid value is from 0 to 15. The default volume level is set by vocMakeConference() function, and 15 is the highest level.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CONFERR	Invalid conference handle.

Example

```
HCONF hConf;
WORD Volume;

Volume = vocGetConfVol(hConf);
vocSetConfVol(hConf, ++Volume);          //Increment volume level in a conference
```

vocSetCSTMask

This function is used to enable the detection of channel status transition, and clear all the pending CST events in the queue.

Syntax

```
WORD vocSetCSTMask(
    HCHN hChn,
    WORD wCstMask
    );
```

Parameters

hChn

Identifies the channel handle.

wCstMask

Specifies the mask bits of CST event. This parameter can be a combination of the following values:

Values	Meaning
CSM_RING	Waits for rings.
CSM_DIGIT	Waits for DTMF.
CSM_SILON	Waits for silence on.
CSM_SILOFF	Waits for silence off.
CSM_ONHOOK	Waits for On-hook.
CSM_OFFHOOK	Waits for Off-hook.
CSM_LCREV	Waits for loop current reversal.
CSM_LCDROP	Waits for loop current drop.
CSM_TONEON	Waits for a user-defined tone detected.
CSM_HANDSET	Waits for a handset status changed, either on-hook or off-hook (only for 4R or 8R card).
CSM_ALL	Waits for all the CST events.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

This function controls the CST event detection. The default CST mask of an opened channel disables all CST events detection.

Example

```
HCHN  hChn;

// Enable the silence-on and silence-off detection.
if (vocSetCSTMask(hChn, CSM_SILON | CSM_SILOFF) != E_OK) {
    // Process error
}
```

vocSetEventCallback

This function redefines the event callback function.

Syntax

```
WORD vocSetEventCallback (
    HCHN hChn,
    LPCBDESC pCallBack
    );
```

Parameters

hChn

Identifies the channel handle.

pCallBack

Points to a CBDESC structure to specify the callback information. If no callback function is required, this value can be zero. The CBDESC structure has the following form:

```
typedef struct tagCBDesc {
  DWORD dwEventCallback ;
  DWORD dwEventCallbackInst;
  DWORD dwEventFlag;
  DWORD dwEventMsg;
  DWORD dwCSTCallback;
  DWORD dwCSTCallbackInst;
  DWORD dwCSTFlag;
  DWORD dwCSTMsg;
} CBDESC;
```

The detail information of this structure is described by ***vocOpenChn()*** function.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

This function is used to redefine the callback mechanism which has already defined by the ***vocOpenChn()*** function.

vocSetGTDMask

This function is used to enable the Global Tone Detection.

Syntax

```
WORD vocSetGTDMask(
    HCHN hChn,
    WORD wGTDMask,
    LPSTR pUserTone,
    WORD wUserToneLen
    );
```

Parameters

hChn

Identifies the channel handle.

wGTDMask

Specifies the mask bits of GTD tone. This parameter can be a combination of the following values:

Value	Meaning
GTM_SIL	Enables the silence detection.
GTM_NONSIL	Enables the non-silence detection.
GTM_HANGUP	Enables the hang-up tone detection.
GTM_LCDROP	Enables the loop current drop detection.
GTM_LCREV	Enables the loop current reversal detection.
GTM_SIT1	Enables the SIT1 tone detection.
GTM_SIT2	Enables the SIT2 tone detection.
GTM_SIT3	Enables the SIT3 tone detection.
GTM_USER	Enables the user-defined tone detection, and the Tone ID is defined by the ***pUserTone*** field.
GTM_ALL	Enables all the GTD detection.

pUserTone

Points to a byte buffer, which specifies the Tone ID (0 – 255) to enable. This parameter is valid if the ***wGTDMask*** field contains GTM_USER, otherwise set this parameter to NULL.

wUserToneLen

Specifies the number of Tone ID defined by the ***pUserTone***. This parameter is valid if the ***wGTDMask*** field contains GTM_USER, otherwise set this parameter to NULL.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

This function is used to control Global Tone Detection. The default GTD mask of an opened channel enables for all GTD detection.

Example:

```
HCHN  hChn;
BYTE ToneID[3];
// Set the Tone ID array to enable
ToneID[0] = 0, ToneID[1] = 1, ToneID[2] = 5;
// Enable the detection of silence-on, hang-up tone and user-defined tone.
if (vocSetGTDMask(hChn, GTM_SIL | GTM_HANGUP|GTM_USER, ToneID, 3) != E_OK) {
    // Process error
}
```

vocSetHook

This function controls the phone line of the specified channel.

Syntax

```
WORD vocSetHook(
    HCHN hChn,
    WORD wSwitch,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

wSwitch

Specifies the I/O control type, this parameter can be one of the following values:

Value	Meaning
IO_HOOK_OFF	To off hook the specified channel.
IO_HOOK_ON	To hang up the specified channel.
IO_HOOK_DUMMY	To support the On-Hook recording function. This function is available only for some voice board.

wMode

Specifies the running mode. This parameter can be one of the following values:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Return Values

Returns E_OK if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_BUSY	Channel is in use.
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_END	End of set-hook function.

Remarks

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
if (vocSetHook(hChn, IO_HOOK_OFF, DM_SYNC) != E_OK) {
    // Process error
}
```

Example 2: Using function in asynchronous model.

```
HCHN hChn;
EVTBLK Event;
if (vocSetHook(hChn, IO_HOOK_OFF, DM_ASYNC) != E_OK) {
    // Process error
}
// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);
```

vocSetVolume

This function controls the output/input volume level of the specified channel.

Syntax

WORD vocSetVolume(
HCHN hChn,
WORD wType,
WORD wVolume
);

Parameters

hChn

Identifies the channel handle.

wType

Specifies the volume type. This parameter can be one of the following values:

Value	Meaning
OUT_VOLUME	Indicates the output volume level for voice playing.
IN_VOLUME	Indicates the input volume level for voice recording.

wVolume

Specifies the volume level to set. The valid value is from 0 to 15. The default volume level is 8, and 15 is the highest level.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Remarks

The default output and input volume level are configured by the DIAG32.EXE. Applications don't change the input volume level if possible.

Example:

```
HCHN  hChn;

// Set the volume level to 8
if (vocSetVolume(hChn, OUT_VOLUME, 8) != E_OK) {
    // Process error
}
```

vocStopChn

This function stops the current active I/O function on the specified channel.

Syntax

```
WORD vocStopChn(
    HCHN hChn,
    WORD wMode
    );
```

Parameters

hChn

Identifies the channel handle.

wMode

Specifies the running mode. This parameter can be one of the following values:

Value	Meaning
DM_SYNC	Sets this option to run function synchronously.
DM_ASYNC	Sets this option to run function asynchronously.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.

Termination Events:

The possible termination events for this function are listed below:

Event	Meaning
EVT_STOP	Channel stopped by ***vocStopChn()*** function.

Remarks

This function can run in synchronous or asynchronous model specified by the ***wMode*** parameter.

Synchronous Model

By default, this function runs synchronously. It will return a zero (E_OK) to indicate function has completed successfully, and the termination event can be retrieved by calling the ***vocGetLastTerm()*** function. Otherwise, It will return a nonzero value for error code.

Asynchronous Model

When this function runs asynchronously. It will return a zero (E_OK) to indicate the function has initiated successfully, and it will generate a termination event after function completed. The possible termination events are listed above. A nonzero value returned by this function indicates an error occurred.

Example

Example 1: Using function in synchronous model.

```
HCHN  hChn;

if (vocStopChn(hChn, DM_SYNC) != E_OK) {
    // Process error
}
```

Example 2: Using function in asynchronous model.

```
HCHN  hChn;
EVTBLK  Event;

if (vocStopChn(hChn, DM_ASYNC) != E_OK) {
    // Process error
}
// Waiting for termination event.
vocWaitEvent(hChn, &Event, WT_INFINITE);
```

vocSwitchFax

This function is used to allocate or free the fax resource and switch the fax daughter-board to the specified voice channel.

Syntax

```
WORD vocSwitchFax(
    HCHN hChn,
    WORD wCmd
    );
```

Parameters

hChn

Identifies the voice channel handle.

wCmd

Specifies the command to allocate the fax resource from the fax daughter-board. This parameter can be one of the following values:

Value	Meaning
1	To allocate a fax resource and switch the fax unit to the specified voice channel.
0	To free a fax resource.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_DEVICE_IN_USE	Fax line was switched to specified voice channel already.

Remarks

This function is used only for the fax daughter board on the voice card. Please be aware that the voice channel must be opened before the fax line switching to. After this function is successfully called, application can use the same channel both for voice and fax processing.

Please note that although there are more than one-fax daughter boards installed in your system, but only one fax unit is working at a time. Application should call ***vocSwitchFax()*** function to allocate the fax resource, and then the fax function will work at the specified voice channel. After the fax function completed, Application should also call ***vocSwitchFax()*** function to free the fax resource, and the fax resource will be available for other channels.

Example

Example 1: Using this function to send a fax.

```
HCHN  hVocChn, hFaxChn;
if (vocOpenChn(&hVocChn, 0, NULL) != E_OK) {…Process error…}
    :
vocSetHook(hVocChn, IO_HOOK_OFF, DM_SYNC);
    :
if (vocSwitchFax(hVocChn, 1) != E_OK) {…Processing Error…};
if (faxOpenChn(&hFaxChn, ANY_CHN, NULL) != E_OK) {…Processing Error…}          //Defined in FAXLIB32
if (faxSend(hFaxChn,…) != E_OK) {…Processing Error…};                          //Defined in FAXLIB32
faxCloseChn(hFaxChn);
vocSwitchFax(hVocChn, 0);
```

Example 2: Using this function to receive a fax.

```
HCHN  hVocChn, hFaxChn;
if (vocOpenChn(&hVocChn, 0, NULL) != E_OK) {…Process error…}
```

```
    :
vocSetHook(hVocChn, IO_HOOK_OFF, DM_SYNC);
    :
if (vocSwitchFax(hVocChn, 1) != E_OK) {..Processing Error..};
if (faxOpenChn(&hFaxChn, ANY_CHN, NULL) != E_OK) {..Processing Error..}        //Defined in FAXLIB32
if (faxReceive(hFaxChn, ...) != E_OK) {..Processing Error..};                  //Defined in FAXLIB32
faxCloseChn(hFaxChn);
vocSwitchFax(hVocChn, 0);
```

Example 3: Using this function to receive a fax with faxWaitRing() function.

The default setting of fax daughter board will not detect the incoming ring signal. In order to enable the ring detection for fax daughter, the ***vocSetChnIO()*** function should be called with **IO_RINGCTRL_ON** parameter.

```
HCHN  hVocChn, hFaxChn;
if (vocOpenChn(&hVocChn, 0, NULL) != E_OK) {..Process error..}
    :
if (vocSetChnIO(hVocChn, IO_RINGCTRL_ON) != E_OK) {..Process error..}          //Enable ring detection for fax
    :
if (vocSwitchFax(hVocChn, 1) != E_OK) {..Processing Error..};
if (faxOpenChn(&hFaxChn, ANY_CHN, NULL) != E_OK) {..Processing Error..}        //Defined in FAXLIB32
if (faxWaitRing(hFaxChn, 3, WT_INFINITE) != E_OK) {..Processing Error..}       //Defined in FAXLIB32;
vocSetHook(hVocChn, IO_HOOK_OFF, DM_SYNC);
if (faxReceive(hFaxChn, ...) != E_OK) {..Processing Error..}                   //Defined in FAXLIB32;
faxCloseChn(hFaxChn);
vocSwitchFax(hVocChn, 0);
    :
```

vocUnlisten

This function disconnects voice receive channel from SCbus. This function disconnects the voice receive (listen) channel on a PLUS-4LVSC board from the SCbus.

Syntax

```
WORD vocUnlisten (
   HCHN hChn
   );
```

Parameters

hChn

Identifies the channel handle.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value. The possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_FUNERR	Function is not supported in current bus configuration.

Remarks

Calling the ***vocListen()*** function to connect to a different SCbus time slot will automatically break an existing connection. Thus, when changing connections, you need not call the ***vocUnlisten()*** function.

Example

```
HCHN hChn;
if (vocUnlisten(hChn) == E_OK) {
    printf(" \n\rVoice channel is no longer listening!" );
}
```

vocWaitConfEvent

This function is used to retrieve the termination event of ***vocMakeConference()*** function for the specified conference handle.

Syntax

```
WORD vocWaitConfEvent(
    HCONF hConf,
    LPWORD pEvent,
    DWORD dwTimeout
    );
```

Parameters

hConf

Identifies the conference handle.

pEvent

Pointers to an buffer in WORD to receive the termination event as follows:

Values	Meaning
EVT_END	Function is terminated by ***vocBreakConference()***.
EVT_GTD	Function is terminated by GTD tone detection. Call ***vocGetConfGTD()*** function to retrieve the source of GTD event.

dwTimeout

Specifies the time-out interval, in miliseconds. The function returns E_TIMEOUT if the interval elapses. If *wTimeout* is INFINITE, the function's time-out interval never elapses.

Return Values

Returns zero (E_OK) if a termination event was received successfully. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CONFERR	Invalid conference handle.
E_TIMEOUT	Time-out interval elapsed.
E_SYSERR	Failed to synchronize.

Example

```
HCONF hConf;
WORD wEvent;

if (vocWaitConfEvent(hConf, &wEvent, WT_INFINITE) == E_OK) {

    /*Process function completion*/
    switch (wEvent) {
    case EVT_END: {..//Process normal ending}
    case EVT_GTD: {..//Process GTD event}
    }
}
```

vocWaitCST

This function is used to monitor the channel status transition.

Syntax

```
WORD vocWaitCST(
    HCHN hChn,
    LPCSTBLK pCst,
    DWORD dwTimeout
    );
```

Parameters

hChn

Identifies the channel handle.

pCst

Points to a CSTBLK structure to receive the channel status information. The CSTBLK structure has the following form:

```
typedef struct tagCstBlk {
  HCHN hChn;
  WORD wStatus;
  DWORD dwData;
  BYTE bReserved[12];
} CSTBLK;
```

The detail description of this structure is listed below:

hChn

Identifies the channel handle.

wStatus

Specifies the channel status. This field can be one of the following values:

Values	Meaning
CST_RING	Ring signal is detected.
CST_DIGIT	A DTMF detected.
CST_SILON	Silence is now on.
CST_SILOFF	Silence is now off.
CST_ONHOOK	On-hook occurred.
CST_OFFHOOK	Off-hook occurred.
CST_LCREV	A loop current reversal detected.
CST_LCDROP	A loop current drop detected.
CST_TONEON	A user-defined tone detected.
CST_HANDSET	A handset status change event detected.

dwData

Contains the data associated with ***wStatus***. This field has a different definition for each CST status:

CST Status	CST Data
CST_RING	Specifies the timer tick value when ring signal occurred.
CST_DIGIT	Specifies the ASCII digit. (0 – 9, *, #, A – D)
CST_SILON	Specifies the interval for non-silence time. (1 ms units)
CST_SILOFF	Specifies the interval for silence time. (1 ms units)
CST_ONHOOK	N/A.
CST_OFFHOOK	N/A.
CST_LCREV	Reverse state. (LC_NORM2REV or LC_REV2NORM)
CST_LCDROP	N/A.
CST_TONEON	Specifies the user-defined Tone ID.
CST_HANDSET	Specifies the handset status, 0 for hang-up and 1 for off-hook.

dwTimeout

Specifies the time-out interval, in milliseconds. The function returns **E_TIMEOUT** if no CST event is present and the interval elapses. If ***dwTimeout*** is zero, the function will return immediately. If ***wTimeout*** is **WT_INFINITE**, the function's time-out interval never elapses.

Return Values

Returns zero (E_OK) if a CST event was received successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_TIMEOUT	Time-out interval elapsed.
E_SYSERR	Failed to synchronize

Remarks

This function is used to monitor channel status. The status transition event can be masked on or off by calling ***vocSetCSTMask()*** function. If the callback model is used for CST events, programmer should call ***vocGetLastCST()*** to get the channel status in callback routine.

Example

```
HCHN  hChn;
CSTBLK  CST;

if (vocSetCSTMask(hChn, CSM_SILON) != E_OK) {
    // Process error
}
if (vocWaitCST(hChn, &CST, WT_INFINITE) != E_OK) {
    // Process error
}
if (CST.wStatus == CST_SILOFF) {
    printf(" \n\rThe interval of silence time is (%lu) ms." , CST.dwData);
}
```

vocWaitEvent

This function is used to retrieve the termination event of the specified channel.

Syntax

```
WORD vocWaitEvent(
    HCHN hChn,
    LPEVTBLK pEvent,
    DWORD dwTimeout
    );
```

Parameters

hChn

Identifies the channel handle.

pEvent

Points to an EVTBLK structure to receive the event information. The EVTBLK structure has the following form:

```
typedef struct tagEvtBlk {
  HCHN hChn;
  WORD wEvent;
  WORD wTermFun;
  BYTE bReserved[12];
} EVTBLK;
```

The detail description of this structure is listed below:

hChn

Identifies the channel handle.

wEvent

Specifies the termination event. This field can be one of the following values:

Values	Meaning
EVT_END	Function is terminated successful.
EVT_ERR	Function is terminated due to an error. Call ***vocGetLastErr()*** function to retrieve the reason of error.
EVT_GTD	GTD tone detected. Call ***vocGetGTD()*** function to retrieve the reason of GTD detection.
EVT_MAXDTMF	The maximum number of digits has received.
EVT_IDDTIME	Inter-digit delay time elapsed.
EVT_MAXTIME	Maximum function time elapsed.
EVT_STOP	Channel stopped by ***vocStopChn()*** function.
EVT_TERMDT	Terminated by input digit. Call ***vocGetTermDT()*** function to get the terminated digit.

wTermFun

Specifies which function is completed. This field can be one of the following values:

Values	Meaning
CBT_PLAY	Indicates the ***vocPlayFile()*** function is completed and a termination event is generated.
CBT_RECORD	Indicates the ***vocRecordFile()*** function is completed and a termination event is generated.
CBT_GETDT	Indicates the ***vocGetDT()*** function is completed and a termination event is generated.
CBT_DIAL	Indicates the ***vocDial()*** function is completed and a termination event is generated.
CBT_PLAYTONE	Indicates the ***vocPlayTone()*** function is completed and a termination event is generated.
CBT_SETHOOK	Indicates the ***vocSetHook()*** function is completed and a termination event is generated.

CBT_FLASHHOOK Indicates the ***vocFlashHook()*** function is completed and a termination event is generated.

dwTimeout

Specifies the time-out interval, in milliseconds. The function returns **E_TIMEOUT** if no termination event is present and the interval elapses. If **d*wTimeout*** is zero, the function will return immediately. If ***wTimeout*** is **WT_INFINITE**, the function's time-out interval never elapses.

Return Values

Returns zero (E_OK) if a termination event was received successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_TIMEOUT	Time-out interval elapsed.
E_SYSERR	Failed to synchronize

Remarks

This function is used to synchronously monitor channel' s status.

Example

```
HCHN  hChn;
EVTBLK  Event;
if (vocWaitEvent(hChn, &Event, WT_INFINITE) != E_OK) {
    // Process error
}
```

vocWaitRing

This function waits for a specified number of rings and sets the channel to on-hook or off-hook after the rings are detected.

Syntax

```
WORD vocWaitRing(
    HCHN hChn,
    WORD wRings,
    WORD wState,
    DWORD dwTimeout
    );
```

Parameters

hChn

Identifies the channel handle.

wRings

Specifies the number of rings to wait.

wState

Specifies the Hook State to set after the number of rings is detected. This parameter can be one of the following values:

Value	Meaning
IO_HOOK_ON	Channel remains on-hook when the number of rings is detected.
IO_HOOK_OFF	Channel gets off-hook when the number of rings is detected.
IO_HOOK_DUMMY	Channel gets a dummy off-hook when the number of rings is detected.

dwTimeout

Specifies the time-out interval, in milliseconds. The function returns **E_TIMEOUT** if no ring signal is detected and the interval elapses. If ***dwTimeout*** is zero, the function will return immediately. If ***wTimeout*** is **WT_INFINITE**, the function's time-out interval never elapses.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_TIMEOUT	Time-out interval elapsed.
E_IOERR	driver IO operation failed.

Example

```
HCHN  hChn;

// Wait forever until 3 rings reached then off-hook
if (vocWaitRing(hChn, 3, IO_HOOK_OFF, WT_INFINITE) != E_OK) {
    // Process error
}
```

vocWaitRingEx

This function waits for a specified number of rings and sets the channel to on-hook or off-hook after the rings are detected. A caller' s phone number is also returned by this function.

Syntax

```
WORD vocWaitRingEx(
    HCHN hChn,
    WORD wRings,
    WORD wState,
    DWORD dwTimeout.
    LPSTR pBuf
    );
```

Parameters

hChn

Identifies the channel handle.

wRings

Specifies the number of rings to wait.

wState

Specifies the Hook State to set after the number of rings is detected. This parameter can be one of the following values:

Value	Meaning
IO_HOOK_ON	Channel remains on-hook when the number of rings is detected.
IO_HOOK_OFF	Channel gets off-hook when the number of rings is detected.
IO_HOOK_DUMMY	Channel gets a dummy off-hook when the number of rings is detected.

dwTimeout

Specifies the time-out interval, in milliseconds. The function returns **E_TIMEOUT** if no ring signal is detected and the interval elapses. If ***dwTimeout*** is zero, the function will return immediately. If ***wTimeout*** is **WT_INFINITE**, the function's time-out interval never elapses.

pBuf

Pointers to a buffer to receive the caller' s phone number. The caller' s phone number is a null-terminated ASCIIZ string. A null string will be returned if no caller' s phone number detected. The maximum length of caller' s phone number is 300 (MAX_CID_LENGTH) bytes.

Return Values

Returns zero (E_OK) if the function was successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
E_CHNERR	Invalid channel handle.
E_TIMEOUT	Time-out interval elapsed.
E_IOERR	driver IO operation failed.

Remarks

The Caller ID feature is not supported on the all voice boards.

Example

```
HCHN  hChn;
char buf[MAX_CID_LENGTH];

// Wait forever until 3 rings reached then off-hook
if (vocWaitRing(hChn, 3, IO_HOOK_OFF, WT_INFINITE, buf) != E_OK) {
    // Process error
}
printf("\n\rCaller ID is %s" , buf);
```

CHNMON32 Program

The channel monitor program CHNMON32.EXE supports a MDI window interface and can display all the channel messages on the individual MDI window. It also can save all the output messages in log files for a long time debug.

If a windows program wants to show its debug messages on the CHNMON32 program, it can call ***monShowCST()*** and ***monShowMSG()*** functions of the ChnMon32.DLL. The function specification of ChnMon32 is described below:

monShowCST

This function shows the CST status on the CHNMON32 program.

Syntax

```
WORD monShowCST (
    WORD wChnID,
    WORD wStatus,
    DWORD dwData
    );
```

Parameters

wChnID

Identifies the channel number. The channel number starts from 0 and can be got by calling ***vocGetChnID()*** function.

wStatus

Specifies the channel status. This parameter can be one of the following values:

Values	Meaning
CST_RING	Ring signal is detected.
CST_DIGIT	A DTMF detected.
CST_ONHOOK	On-hook occurred.
CST_OFFHOOK	Off-hook occurred.

dwData

Contains the data associated with ***wStatus***. This parameter has a different definition for each CST status:

CST Status	CST Data
CST_RING	0
CST_DIGIT	Specifies the ASCII digit. (0 – 9, *, #, A – D)
CST_ONHOOK	Reserved to 0.
CST_OFFHOOK	Reserved to 0.

Return Values

Returns zero (MONMSG_OK) if the function was successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
MONMSG_CHNID_ERR	Invalid channel number.
MONMSG_FILE_MAP_ERR	Unable to create map file.

Example

```
#include " ChnMon32.h"

HCHN hChn;
WORD wChnID;

WChnID = vocGetChnID(hChn);
If (monShowCST(wChnID, CST_ONHOOK, NULL) != MONMSG_OK) {
    // Process error
}
```

monShowMSG

This function shows the message on the CHNMON32 program.

Syntax

```
WORD monShowMSG (
    WORD wChnID,
    LPSTR lpFmt,
    ...
    );
```

Parameters

wChnID

Identifies the channel number. The channel number starts from 0 and can be got by calling ***vocGetChnID()*** function.

lpFmt

Points to a null-terminated string that contains the format-control specifications. In addition to ordinary ASCII characters, a format specification for each argument appears in this string. For more information about the format specification, see the ***wsprintf()*** function of Windows API.

…

Specifies one or more optional arguments. The number and type of argument parameters depend on the corresponding format-control specifications in the ***lpFmt*** parameter.

Return Values

Returns zero (MONMSG_OK) if the function was successful. Otherwise, it returns a nonzero value and the possible error returns are:

Value	Meaning
MONMSG_CHNID_ERR	Invalid channel number.
MONMSG_FILE_MAP_ERR	Unable to create map file.
MONMSG_TEXT_ERR	Invalid text string.

Example

```
#include " ChnMon32.h"

HCHN hChn;
WORD wChnID;

WChnID = vocGetChnID(hChn);
monShowMSG(wChnID, " Channel ID = %u,  Playing voce file(%s)..", wChnID, " voice.wav" );
    // Process error
}
```

Application Notes

- Caller ID
- ADSI (Analog Display Services Interface)
- Voice Logging System
- SC Bus Application

Caller ID

Overview

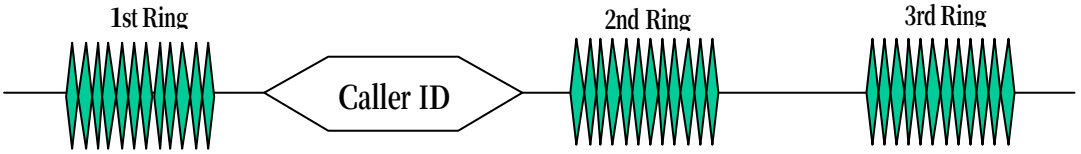
Caller Identification (Caller ID) is a feature that enables the called lines to receive the caller's phone number, possibly date, time, name of caller, and other information about the call. There are two protocols to transmit the Call ID information, one is FSK (Frequency Shift Keying) and another is DTMF signal.

FSK

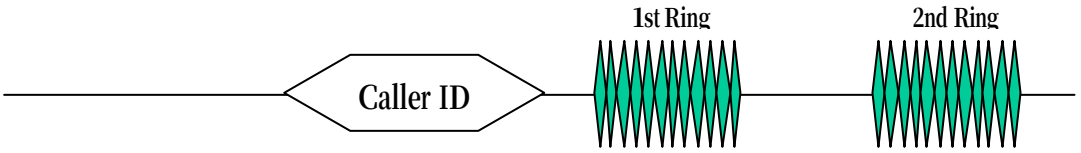
The FSK feature provided by Plus-series voice boards is transmitted at 1200 baud, and compliant to the V.23 standard characteristics. It supports all countries that use the Bellcore CLASS specification. The Caller ID formats currently supported are:

- Custom Local Area Signaling Services (CLASS) is a standard published by Bellcore.
- Analog Calling Line Identity Presentation (ACLIP) is a standard used in Singapore.
- Calling Line Identity Presentation (CLIP) is a standard used in the Unit Kingdom.

For CLASS and ACLIP, the Caller ID information received from the CO (Central Office) line is between the first ring and second ring signal.



For CLIP, the Caller ID information received from the CO (Central Office) line is before the first ring signal.



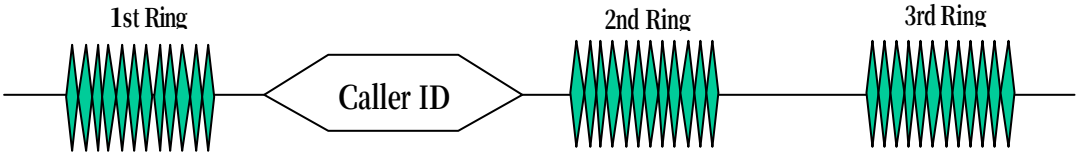
The Caller ID information for CLASS and ACLIP contains two format types:

- **Single Data Message (SDM)** format, which includes the following information:
  - SDM format type
  - Calling line Identity (CLI) (i.e. Calling phone number)
  - Date
  - Time
- **Multiple Data Message (MDM**) format, which includes the following information:
  - MDM format type
  - Calling line Identity (CLI) (i.e. Calling phone number)
  - Date
  - Time
  - Calling party name (CPN)
  - Call Type
  - First Called Line Identity
  - Type of forwarded call
  - Reason for absence of Calling Line Identity
  - Reason for absence of Calling Party Name

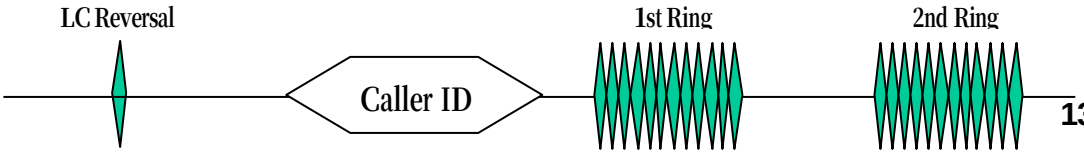
DTMF

The Caller ID transmitted by DTMF signal has two signal sequences:

The Caller ID is transmitted between the first ring and the second ring signal. The signal sequence is as follows:



The Caller ID is following a Loop Current Reversal signal, and the ring signals are following the Caller ID. The signal sequence is as follows:



The Caller ID transmitted by DTMF signal has the following format:

<D> CallerPhoneNumber <C>

If the caller number is 2219-5499, the received Caller ID string will be D22195499C. The digit 'D' indicates the starting character of the caller phone number. The 'C' digit indicates the ending character of the caller phone number.

Enabling the Caller ID feature

Applications can enable the Caller ID feature on the specified channel to process Caller ID information as it is received with an incoming call. For applications, after a channel is opened via ***vocOpenChn()*** function, the default Caller ID feature is disabled. Applications should call the ***vocSetChnParam()*** function with **CP_CALLERID** parameter to enable the Caller ID feature on the specified channel. The more detail usage of ***vocSetChnParam()*** function, please refer to the functional description. Driver cannot handle the FSK and DTMF detections at the same time, applications only can choose one detection method for each channel.

The ***vocWaitRingEx()*** function allows applications to wait for a specified number of rings and returns the caller's phone number.

The ***vocGetCallerID()*** function allows applications to retrieve the Caller ID information if available.

For those applications which use **FSK** method, please note that the seizure count of CallID signal could be adjusted by the **DSPCMD.INI** under Windows directory as follows:
[VOCLIB]
CIDSeizure=XXX, the default setting is 280. Try to lower value if your application did not receive caller ID. For example, you can try if 280, 270, 260... works.

Example

The example program code listed below describes how to enable the Caller ID feature and retrieve the Caller ID string on the specified channel:

```
HCHN hChn;
char CallID[MAX_CID_LENGTH];
```

Caller ID

```
// Open a channel
vocOpenChn(&hChn, ANY_CHN, NULL);

// Enable the caller ID feature on the specified channel
// CPX_??? = CPX_FSK for FSK detection
// CPX_??? = CPX_DTMF for DTMF detection
vocSetChnParam(hChn, CP_CALLERID, CPX_???);

//WaitRing:
// Wait for incoming call and off-hook the channel if number of ring reached
vocWaitRingEx(hChn, 2, IO_HOOK_OFF, WT_INFINITE, CallID);

// Caller ID string is available in CallID buffer
printf(" \n\rCaller ID = %s" , CallID);

:
:
// Playing voice prompt
vocPlayFile(hChn, " greet.wav" , 0, DT_ALL, DM_SYNC);
:
:
:
// Hang up the channel
vocSetHook(hChn, IO_HOOK_ON, DM_SYNC);
// Go back and wait for next call.
goto WaitRing;
/
```

ADSI (Analog Display Services Interface)

Overview

To be an IVR Server for ADSI applications, the Plus-series voice boards contains the following features:

- Voice Prompt
- Detects DTMF signals for users' inputs and acknowledgment (ACK) tone
- Generates CPE Alert Signal (CAS) tone
- Demodulates 1200-baud FSK (V.23) modem data
- Transmits 1200-baud FSK (V.23) modem data

The modem-like protocol based on Frequency Shift Keying (FSK) is half-duplex. Applications can transmit or receive an ASCII string or binary data directly. The channel seizure of FSK frame and CAS tone are programmable to meet the communication protocol with remote devices. The available function calls for ADSI handshake are listed below:

adsiCAS()	Generates a CAS tone to remote devices and waits for an ACK tone.
adsiSetParam()	Changes channel's seizure signal and CAS tone settings.
adsiRecvFrame()	Receives a V.23 FSK frame.
adsiXmitFrame()	Transmits a V.23 FSK frame.

The following function calls will possibly use for an ADSI server application:

vocWaitRing()	Waiting for incoming calls.
vocSetHook()	Controls the phone line.
vocGetDT()	Collects digits from the channel's DTMF queue.
vocClearDT()	Clears the channel's DTMF queue.
vocPlayFile()	Plays back the voice file(s).
vocPlayTone()	Generates a single or dual frequency tone.

Frame Format

A V.23 FSK frame has the following format:

Seizure Signal	Mark Signal	Message
10*n bits	m bits	x bytes

Seizure Signal The Seizure Signal is used to synchronize the FSK data and indicates the starting signal of a FSK frame. The seizure signal is a series of alternating 0 and 1 bits. For an ADSI frame protocol, the default number of alternating bits is 50 (i.e. 25 bit 1 and 25 bits 0). It can be programmable by calling the ***adsiSetParam()*** function. To prevent from using the seizue signal, set the alternating bits to zero.

Mark Signal The Mark Signal is used to separate the seizure signal and message block. The mark signal is a series of at least 55 bits 1 (stop bit) to indicate no data transmission.

Message The Message is the real data that sender wants to transmit. The definition for this message block is depending on the requirements of application. In general, the Message block can be divided into the following fields:

Message Type	Message Length	Message Data	Check Sum

Message Type

The Message Type is a single binary byte. The value depends on the application. (To prevent from a conflict with seizure signal, don't set value 0x55 to the message type).

Message Length

The Message Length is a single binary byte (or word) indicating the number of bytes in the Message block, excluding the Message Type, Message Length, and CheckSum bytes. (i.e. The total length of Message Data)

Message Data

The Message Data consists of between 0 and 255 bytes, according to the Message Length field. Any 8-bit value may be sent, depending on the requirements of the application.

CheckSum

The CheckSum consists of a single byte equal to the two's complement sum of all bytes starting from the **Message Type** up to the end of the message block. Carry from the most significant bit is ignored. The receiver must

compute the 8-bit sum of all Message block bytes and **CheckSum** byte, and the result must be zero.

Note: The Seizure* and *Mark* Signals *will automatically be added to the ADSI frame for the transmission function. These signals will also be removed from the ADSI frame for the receiving function. If the seizure signal is not applied for the ADSI protocol, applications can set the alternating bits of seizure signal to zero for receiving and transmission functions.

FSK Modulation and Demodulation

The standard FSK transmission frequencies are listed below:

Modem Standard	Carrier (Hz)	1 (Mark) (Hz)	0 (Space) (Hz)
V.23	1700	1300	2100
Bell 202	1700	1200	2200

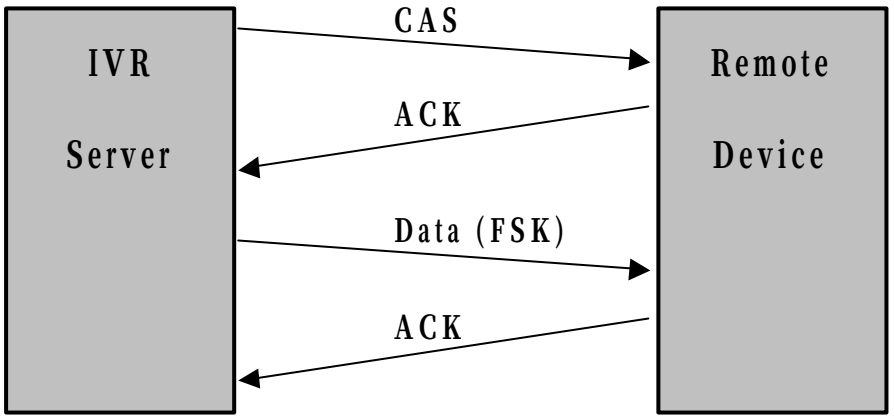
The Plus-series voice boards are capable of decoding data transmitted in either V.23 or Bell 202 1200 baud standards with 1 start, 8 data and 1 stop bit per symbol. The FSK modem data is decoded using digitial quadrature demodulation techniques. In order to detect the different FSK frame, the channel seizure of receiving frame is programmable.

The FSK transmission frequencies for Plus-series voice boards are using 1275Hz and 2125Hz. Both the V23 and Bell 202 standards are close enough to demodulate these frequencies no matter which standard is actually being used. The channel seizure for a transmission frame is programmable, and the mark signal consists of 60 bits 1.

Example

The example program code listed below describes how to communicate with the remote device using the following protocol:

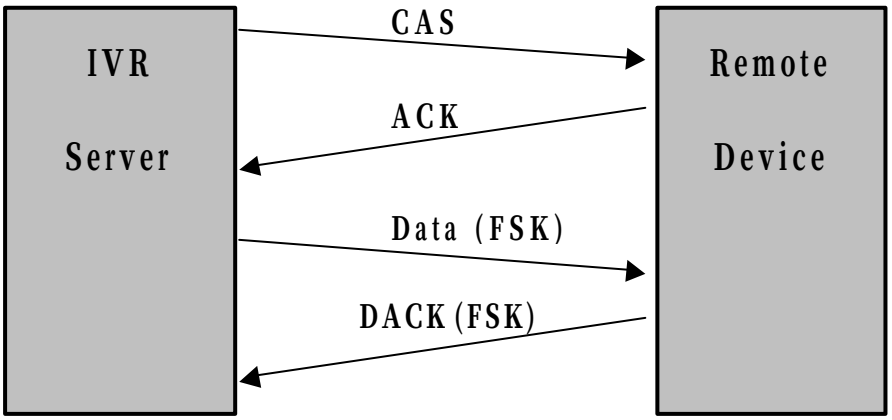
ADSI




```
#define DATA_LEN            20

typedef struct tagXmitFrame {
    BYTE bType;
    BYTE bLength;
    BYTE bData[DATA_LEN];
    BYTE bCS;
    } XMITFRAME;

HCHN hChn;
XMITFRAME Frame;

// Open a channel
vocOpenChn(&hChn, ANY_CHN, NULL);

//WaitRing:
// Wait for incoming call, and off-hook the channel if number of ring reached
vocWaitRing(hChn, 1, IO_HOOK_OFF, WT_INFINITE);

// Generates a CAS and waits for ACK tone (DTMF digit "A")
if (adsiCAS(hChn, 3000, DT_A, DM_SYNC) != E_OK) {
    //Process error;
    }
switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_MAXTIME:
        // Time-out, no ACK tone detected.
        goto Error;
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }

//Prepare Frame
Frame.bType =0;
Frame.bLength = DATA_LEN;
// Fill frame data into Frame.bData
    :
Frame.bCS = adsiCheckSum(&Frame, sizeof(XMITFRAME)-1);  // -1 is used to exclude the CheckSum byte.
// Frame is ready to trabnsmit.

// Transmit a frame.
```

```
if (adsiXmitFrame(hChn, &Frame, sizeof(XMITFRAME), DM_SYNC) != E_OK) {
    //Process error;
    }
switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }

// Waiting for ACK tone (DTMF digit "A");
if (adsiCAS(hChn, 3000, DT_A, DM_NOCAS|DM_SYNC) != E_OK) {
    //Process error;
    }
switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_MAXTIME:
        // Time-out, no ACK tone detected.
        goto Error;
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }

:
:
// Hang up the channel
vocSetHook(hChn, IO_HOOK_ON, DM_SYNC);
// Go back and wait for next call.
goto WaitRing;
```

The example program code listed below describes how to communicate with the remote device using the following protocol:



```
#define DATA_LEN              20

typedef struct tagXmitFrame {
    BYTE bType;
    BYTE bLength;
    BYTE bData[DATA_LEN];
    BYTE bCS;
    } XMITFRAME;

typedef struct tagRecvFrame {
    BYTE bType;
    BYTE nCC;
    BYTE bLength;
    BYTE bData[DATA_LEN];
    BYTE bCS;
    } RECVFRAME;

HCHN hChn;
XMITFRAME XmitFrame;
RECVFRAME RecvFrame;
WORD wMsgSize;

// Open a channel
vocOpenChn(&hChn, ANY_CHN, NULL);

//WaitRing:
// Wait for incoming call, and off-hook the channel if number of ring reached
vocWaitRing(hChn, 1, IO_HOOK_OFF, WT_INFINITE);

// Generates a CAS and waits for ACK tone (DTMF digit " A" )
if (adsiCAS(hChn, 3000, DT_A, DM_SYNC) != E_OK) {
    //Process error;
    }
switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_MAXTIME:
        // Time-out, no ACK tone detected.
        goto Error;
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
        goto Error;
        break;
    }


//Prepare Frame
XmitFrame.bType =0;
XmitFrame.bLength = DATA_LEN;
// Fill frame data into XmitFrame.bData
    :
XmitFrame.bCS = adsiCheckSum(&XmitFrame, sizeof(XMITFRAME)-1); // -1 is used to exclude the CheckSum byte.
// Frame is ready to trabnsmit.

// Transmit a frame.
if (adsiXmitFrame(hChn, &XmitFrame, sizeof(XMITFRAME), DM_SYNC) != E_OK) {
    //Process error;
    }
switch (vocGetLastTerm()) {
    case EVT_END:
        break;
    case EVT_GTD:
        // Caller has hanged up the line already.
        goto HangUp;
        break;
    default:
```

```
            goto Error;
            break;
        }

// Receive a frame
if (adsiRecvFrame(hChn, &RecvFrame, sizeof(RECVFRAME), &wMsgSize, 5000, DM_SYNC) != E_OK) {
        //Process error;
        }
switch (vocGetLastTerm()) {
        case EVT_END:
            break;
        case EVT_ERR:
            // Check sum error.
            break;
        case EVT_GTD:
            // Caller has hanged up the line already.
            goto HangUp;
            break;
        default:
            goto Error;
            break;
        }

:
:
:
// Hang up the channel
vocSetHook(hChn, IO_HOOK_ON, DM_SYNC);
// Go back and wait for next call.
goto WaitRing;
```

Frequently Asked Questions

1. How to make sure voice card is working after installed?
   - Run program from "Start"->"Programs"->"V-Link32 Development Kit"->"Line Detector". If **"Waveform driver is not installed"** message appears, the installation was failed. If Line Detector program could be executed, then the card is working.

2. How to fix the problem of "**Waveform driver is not installed**"?
   - Check if hardware's setting matches with software's settings:
     - ✓ Hardware settings: read "Installation Manual" to find out the jumper settings for IRQ, IO port and Shared memory address. The factory settings are IRQ(5), IO(360) and Shared memory(D000).
     - ✓ Software settings:
       - ✧ For NT user: Run program from "Start"->"Programs"->"V-Link32 Development Kit"->"Voice Settings" to configure the hardware settings.
       - ✧ For Win98 user: Run program from "Start"->"Settings"->"Control Panel"->"System"->"Device Manager". Look at "Audio voice cards" class and select "ADmore voice card". Double click to view the resources.
   - Check card's resource conflicts with other devices:
     - ✓ For NT user: Run program from "Start"->"Programs"->"Administrative Tools (Common)"->"Windows NT Diagnostics". Select "Resources" tab to overview all the resources including IRQ, IO Ports and Memory are occupied by existing devices in the system. The device name for our voice card is "TPLUS". You can look at if the resources for TPLUS is conflicting with others.
     - ✓ For Win98 user: Run program from "Start"->"Settings"->"Control Panel"->"System"->"Device Manager". Look at "Audio voice cards" class and select "ADmore voice card". Double click to view the resources status. If any IRQ, IO and Memory conflicts, there will be certain error sign showing.
     - ✓ If voice card resources does conflicts, please change both software and hardware settings for voice card, or modify the resources for conflicting device.
   - Make sure IRQ reserved for ISA card: when booting the system, enter system BIOS, choose "PNP/PCI setting". If your voice card is set at IRQ5, then the IRQ 5 setting in BIOS should be reserved for "ISA card".

3. If V-Link32 is workable on Windows/95?
   - The answer is NO because of different device driver support. If user wishes to use Win95 platform, please use either AG16 or 16 Bits development kit.

4. What developing language could be used for V-Link32?
   - V-Link32 API is the same as Windows API which is written in Dynamic Link Library. The most common used languages are Microsoft C++, Borland C++ Builder, Delphi and Visual Basic.

5. How to use V-Link32 with Microsoft C++?
   - Note 1: refer to "VC" sample directory which is located under V-Link32 installed directory.
   - Note 2: in code module, include "VOCLIB32.H" which is located in "INC" directory.
   - Note 3: before start project link, add "VOCLIB32.LIB" which is located in "LIB" directory.

6. How to use V-Link32 with BCB(Borland C Builder)?
   - Note 1: refer to "BCB" sample directory which is located under V-Link32 installed directory.
   - Note 2: in code module, include "VOCLIB32.H" which is located in "INC" directory.
   - Note 3: before start project link, add "VOCLIB32.LIB" which is located in "LIB" directory.

7. How to use V-Link32 with VB(Visual Basic)?
- Note 1: refer to "VB" sample directory which is located under V-Link32 installed directory.
- Note 2: Add both "VOCLIB32.BAS" and "COMMON.BAS" into projects, which is located under "LIB" directory.

8. How to use V-Link32 with Delphi?
- Note 1: refer to "Delphi" sample directory which is located under V-Link32 installed directory.
- Note 2: Add "VOCLIB32.PAS" into projects, which is located under "LIB" directory.

9. What is the first step to write V-Link32 application program?
- Before any operation to the channel on the voice card, you must call vocInitDriver() to initialize device driver, then call vocOpenChn() to get a channel handle for further operation. If the channel is no need to use, call vocCloseChn() to close and free channel resources.

10. What is the difference using "Synchronous" and "Asynchronous" program model?
- If application program specifies "Synchronous" model, then calling into V-Link32 API will block the execution until completes its function. Vice versa, "Asynchronous" model, the API will return immediately after starting its function by creating a thread running in the background. For BCB and VC project, it is free to use either synchronous or asynchronous mode, because in C++, the application is able to create thread for their need. But for VB project, the most common implementation is to use timer interval to do their tasks, therefore, the "Asynchronous" model is the only way to do.

11. What is event management for during V-Link32 API call?
- Event is as communication data between application program and V-Link32 inside. Whenever application wants to know if the asynchronous function is done, the application can call vocWaitEvent() to get function result, such as EVT_END means function completes, EVT_STOP means function was stop by user, EVT_TERMDT means function is stop by input digit, or vocWaitEvent() returns E_TIMEOUT means the function is still going on, but the waiting time is up.

12. What voice file format are supported for voice card?
- 6K and 8K sampling rate, and u-Law PCM, ADPCM and Windows PCM, 8Bit and mono channel.

13. What is the smallest file size to record voice into a file?
- 6K sampling rate, 4 Bits ADPCM, i.e. 3000 bytes per second. You can specify VM_ADPCM+VM_SR6 as wMode parameter in vocRecordFile().

14. How to play a voice file, and is terminated by input digit(s)?
- Using vocPlayFile(hChn, "Greet.wav", 0, DT_9+DT_*....) means "Greet.wav" file will be played, and terminated by either "9" or "*" digit is pressed. If the application program requires to know which digit was pressed("9" or "*"), just call vocGetTermDT() to retrieve exact pressed digit.

15. How to detect Hang-up tone or a specific tone?
- To use GTD(Global Tone Detection) which is set in "Line Detector Program(Diag32.exe)", you can specify the tone frequency, silent off time period, silent on time period, and Tone ID. Besides, you can learn the tone setting in Diag32.exe by following its recommended steps. Once the tone characteristic is set and happens, the V-Link32 API will return EVT_GTD to let application know specific tone (by tone ID) was detected. Usually, EVT_GTD is often used as hang up tone.

16. How to generate a tone?
- To use vocPlayTone() to generate a tone, you can specify tone frequency (from 0~4KHz) and also duration for the time period.

17. How to trun on speaker to listen to channel status?
- Use vocSetChnIO(hChn, IO_SPK_ON).

18. How to receive DTMF digit(s)? What is the difference for vocGetDT() and vocReadDT()?
- After channel is off hook, voice driver is keep on monitoring DTMF digit, and place it into a internal queue if any. Using vocGetDT(), you can specify the condition of maximum time to wait, maximum digits to wait and also certain digit(s) to wait, and return different event to let application know what happened, i.e. EVT_MAXTIME, EVT_MAXDTMF,and EVT_TERMDT. Regarding vocReadDT(), just simply check if the DTMF queue contains digit(s). If yes, pick one out for each function call, or return ZERO if none exists in the queue.

19. How to know the dialing result?
- Application call vocDial() to do outbound call, and after vocDial() function is completed, vocGetCAR() function will get you know the result, such as "No answer", "Busy", "Connected" or "No Dial Tone".

20. What is call progress monitor?
- Let's analyze the process in vocDial() which initially check if the channel is off hook or not, if not, get off hook first. Then starts to monitor the line signal including, it may appear ring back tone, busy tone, line noise(certain silent off), or human voice if people pick up the line. For example, if voice card receives cadenced ring back for a time period, then this situation will be judged as "No answer". The same implementation happens on "Busy" result. For "No Dial Tone" is because voice card is not able to detect predefined dial tone. You may ask what the criteria to detect human voice is? The characteristic for human voice is that we don't have certain fixed frequency, not like busy tone owning 480/620Hz characteristic. Therefore, to detect dial tone, busy tone, or ring back tone is the key factor to successfully know the dial out result. To let voice card very clearly knows what the busy tone, dial tone or ring back tone is, you can specify those settings on the "Call Progress Monitor" of "Diag32.exe". Of course, we also provide the learning way to know those settings. Besides, for simple usage, we also provide "Intelligent Call Progress Monitor" which collects most common used settings for busy tone and ring back tone, so you don't have to specify those settings unless your telephony system is not suitable for those settings.

21. What is CST? What is the CST for?
- CST means (Channel Status Transition) which provides a way to let application know current channel status change. The status change includes On/Off hook, Ring, DTMF, Silent On/Off time, Loop Current reversal/drop(for off-hook in certain telephony system) , Specific tone happened. One thing to note that before use CST, you have to open each mask for each of them by calling vocSetCSTMask(), and then calling vocWaitCST() to get above status change.